Ahead-of-Time Algebraic Compilation for Safety-Critical Java

James Baxter

PhD

University of York

Computer Science

July 2018

Abstract

In recent years Java has been increasingly considered as a language for safety-critical embedded systems. However, some features of Java are unsuitable for such systems. This has resulted in the creation of Safety-Critical Java (SCJ), which facilitates the development of certifiable real-time and embedded Java programs. SCJ uses different scheduling and memory management models to standard Java, so it requires a specialised virtual machine (SCJVM). A common approach is to compile Java bytecode program to a native language, usually C, ahead-of-time for greater performance on low-resource embedded systems.

Given the safety-critical nature of the applications, it must be ensured that the virtual machine is correct. However, so far, formal verification has not been applied to any SCJVM. This thesis contributes to the formal verification of SCJVMs that utilise ahead-of-time compilation by presenting a verification of compilation from Java bytecode to C.

The approach we adopt is an adaptation of the algebraic approach developed by Sampaio and Hoare. We start with a formal specification of an SCJVM executing the bytecodes of a program, and transform it, through the application of proven compilation rules, to a representation of the target C code. Thus, our contributions are a formal specification of an SCJVM, a set of compilation rules with proofs, and a strategy for applying those compilation rules.

Our compilation strategy can be used as the basis for an implementation of an ahead-of-time compiling SCJVM, or verification of an existing implementation. Additionally, our formal model of an SCJVM may be used as a specification for creating an interpreting SCJVM. To ensure the applicability of our results, we base our work on icecap, the only currently available SCJVM that is open source and up-to-date with the SCJ standard.

Contents

List of Tables

List of Figures

Acknowledgements

Firstly, I would like to thank my supervisor, Ana Cavalcanti, for all her advice on the work and comments on this thesis as it developed, and for always replying to emails promptly when I required help. My thanks also go to Andy Wellings, for the clarifications of the Safety Critical Java specification that he offered and for bringing the issues I raised to the attention of the expert group developing that specification. I would also like to thank my assessor, Colin Runciman, for his comments on drafts of the thesis and suggestions of related literature.

Leo Freitas gave great help with the use of Z/EVES and Community Z Tools, and I am thankful to him for the time he took to explain these things. My thanks also go to the authors of the icecap HVM for their help in using it. I am thankful to Augusto Sampaio for his feedback on my use of the algebraic approach. I would also like to thank Pedro Ribeiro for his useful advice on the format of this thesis, and Alvaro Miyazawa for his help in putting technical reports online. Heartfelt thanks go to Simon Foster for all his friendship throughout this PhD, for teaching me to use the Isabelle proof assistant, and for introducing me to the area of formal methods, without which I may never have done this PhD at all.

This work has been funded by an EPSRC studentship, and I am grateful to EPSRC for their provision of the funding and to the University for deciding to allocate one of the studentships to me.

My parents have given me a great amount of support and encouragement over the years, and I am particularly grateful to them for how they have supported me while I have been busy writing this thesis. I would particularly like to thank my mother for proofreading this thesis. My thanks also go to all my friends for their support, particularly to the members of Trinity Church York and Wentworth Christian Union for all their support, encouragement, and prayers.

Finally, as a Christian, I acknowledge that all that happens is in the hands of God. I am enormously thankful to him for all he has done in bringing me safely to this point. Therefore, I commit this thesis to his sovereign care and ask that he alone may be glorified in all things.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as references. The following material, presented in this thesis, has been previously published:

[1] James Baxter and Ana Cavalcanti. “Algebraic Compilation of Safety-Critical Java Bytecode”. In: *Integrated Formal Methods: 13th International Conference, iFM 2017*. Ed. by Nadia Polikarpova and Steve Schneider. Springer International Publishing, 2017, pp. 161–176. ISBN: 978-3-319-66845-1. DOI: `10.1007/978-3-319-66845-1_11`.

[2] James Baxter, Ana Cavalcanti, Andy Wellings, and Leo Freitas. “Safety-Critical Java Virtual Machine Services”. In: *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES ’15. New York, NY, USA: ACM, 2015, 7:1–7:10. ISBN: 978-1-4503-3644-4. DOI: `10.1145/2822304.2822307`.

The following published material is not directly described in this thesis, but provides evidence of the context in which our work can be applied. We discuss it here.

[3] Leo Freitas, James Baxter, Ana Cavalcanti, and Andy Wellings. “Modelling and Verifying a Priority Scheduler for an SCJ Runtime Environment”. In: *Integrated Formal Methods: 12th International Conference, iFM 2016*. Ed. by Erika Ábrahám and Marieke Huisman. Cham: Springer International Publishing, 2016, pp. 63–78. ISBN: 978-3-319-33693-0. DOI: `10.1007/978-3-319-33693-0_5`.

Chapter 1

Introduction

This chapter begins by explaining the motivation for the work described in this dissertation. Afterwards, the objectives of the work, which come from the motivation, are described and then the approach taken in the work is discussed. Finally, the structure of the remainder of this dissertation is described.

1.1 Motivation

Since its release in 1995, the Java programming language [41] has increased in popularity and is now in use on many platforms. This popularity means that Java has been used in a wide variety of areas including desktop applications, on the internet in the form of Java applets, on smartcards [27] and on mobile devices [90]. Several languages derived from Java have also been created, including Scala [36] and Ceylon [96], as well as older variants of Java such as MultiJava [28] and Pizza [87], which have in turn contributed to the development of Java. Scala adds functional programming features to Java, some of which have been incorporated into Java 8. Ceylon extends Java's type system with features such as union types, allowing some common Java errors to be checked at compile time through the type system.

One use of Java that is of particular interest is in embedded systems. While early versions of Java were developed for programming embedded systems, particularly TV set-top boxes, the technology was not well received. It was only in the growing sector of the internet that Java initially found a market [49]. However, it was soon realised that the portability, modularity, safety and security benefits of Java could be of great use in embedded systems [83]. This required the creation of specialised Java virtual machines as the standard JVM is too large for most embedded systems. Much research has gone into making smaller and smaller virtual machines to widen the range of devices that Java can be used on [22, 115].

Many embedded systems are also real-time systems, and features of Java such as the garbage collector and the concurrency model make it unsuitable for real-time systems, for which strict guarantees about timing properties must be made. To address this issue the Real-Time Specification for Java (RTSJ) [40] was created. RTSJ extends Java with a scoped memory model and a more predictable scheduling system. It also allows for garbage-collected heaps alongside the scoped memory model, using more predictable real-time garbage collectors [104, 107, 117].

While RTSJ addresses real-time requirements of embedded systems, many embedded systems are also safety-critical. For these conformance to certain standards, such as DO-178C and

ISO 26262, is required. To support the development of safety-critical programs that meet these requirements in Java, the Safety-Critical Java (SCJ) specification [67] has been created. SCJ is a subset of RTSJ that removes the features that cannot be easily statically reasoned about, which means that features such as the garbage-collected heap and dynamic class loading are absent from SCJ. This facilitates creation of SCJ programs that fulfil formal specifications; indeed work has already been done on developing correct SCJ programs from formal specifications [25, 26].

On the other hand, even if it can be shown that SCJ programs are correct, it must still be ensured those programs are executed correctly. In the case of Java-like languages, this generally means ensuring the Java compiler and Java Virtual Machine (JVM) are correct.

Work has been done on modelling virtual machines for Java, and on the formal correctness of compilers targeting those virtual machines. Some of the most complete work in that area was by Stärk, Schmid and Börger [112], who presented a model of the full Java language and virtual machine, along with a formally verified compiler, although for an older version of Java than is current. Other work has also been done on modelling the JVM and Java compilation using refinement techniques [35]. Additionally there has been work considering machine checked models of Java virtual machines and compilers [66, 85, 113]. Work has also been done on the semantics of Java bytecode and verification of standard JVMs [16, 50].

However, SCJ has a number of differences from standard Java. Firstly, as already indicated the SCJ memory model is rather different to the standard Java memory model, abandoning the garbage collector in favour of a scoped memory model. Garbage collection is less predictable and often quite complex, and so difficult to reason about and unsuitable for some of the strictest certifiability requirements of safety-critical systems. By contrast, the scoped memory model provides greater predictability on when memory is freed. Similarly, the SCJ approach to scheduling differs from that of standard Java, using a preemptive priority scheduling approach rather than the unpredictable scheduling of standard Java threads. These differences of SCJ from standard Java mean that the standard JVM is not suitable for running SCJ programs. A specialised virtual machine is required.

In the case of virtual machines for embedded systems, the priorities are usually size and speed, which generally result in machines that are hard to verify. Moreover, virtual machines that rely on interpreting bytecode are unsuitable for real-time embedded systems as they are likely to be slower. An alternative method to run a Java program is to compile it to native code and some authors have suggested doing so either directly [109] or via C [116]. There are several virtual machines that take this approach including Fiji VM [93], icecap HVM [110] and OVM [4]. This allows correct running of an otherwise correct SCJ program to be viewed as a compiler verification problem.

There has been much research into compiler correctness. Much of the work follows a commuting diagram approach, in which the compilation is shown to be consistent with transformation between the semantics of the source and target languages [80, 114]. This approach is apparent in much of the early work such as that of McCarthy and Painter [75], as well as in more recent work such as the CompCert project [59, 60]. There has also been work that follows this approach and employs automated theorem provers [54, 77, 85]. They provide additional certainty that the proof is correct and can also provide code generation facilities to allow creation of a working compiler.

An alternative is the algebraic approach to compiler verification [44, 101], based on modelling compilation using refinement calculi [7, 79, 81]. This approach appears to be less commonly used but has been applied to Java [34, 35] and hardware description languages [91, 92]. It is also quite amenable to automation as it relies on refinement laws that can be applied by a term rewriting system.

There is a clear need for formal verification of SCJ virtual machines (SCJVMs) due to the safety-critical nature of the systems involved and the fact that safety standards such as DO-178C require it at the highest safety levels. However, there appears to be little work done in that area and, as far as we know, no SCJVM has been formally verified.

1.2 Objectives

Our objective is to develop an approach to verification of an SCJ virtual machine that allows the production and verification of correct SCJ virtual machines. Although the actual creation and verification of such machines is outside the scope of our work, we provide the following resources for developers and verifiers:

- a specification of the requirements of an SCJ virtual machine,
- a formal model of the virtual machine specification,
- a compilation strategy from Java bytecode to native C code, and
- proofs for validation of the formal model and verification of the compilation strategy.

We follow the design of existing SCJVMs to ensure that our work is of practical relevance to the SCJ community. We particularly focus on the icecap HVM [110], as that is the only publicly-available SCJVM that is up-to-date with the SCJ specification. Where there are ambiguities or concerns regarding the description of the virtual machine in the SCJ standard, we take the icecap implementation as a reference to define the requirements and formal model for an SCJVM. In addition, the native C code generated by our formal compilation strategy is very close to that actually produced by icecap.

Our results can be used to aid the development and verification of an SCJVM in several different ways. The informal specification provides a reference for the requirements of an SCJVM, while the formal model can be used to prove correctness of an implementation. The formal model could also be used to create a correct-by-construction virtual machine via refinement steps. Similarly, the specification of the compilation strategy can be used to translate SCJ bytecode to equivalent C code, which may be used to add compilation facilities to an SCJVM. The proofs give further assurance of the correctness of the model and compilation strategy. Additionally, the mechanisation can better facilitate the use of the other components of the work.

1.2.1 SCJ Virtual Machine Specification

The first component required is a specification of the requirements for an SCJ virtual machine. This specification shapes the rest of the work and there is at present no clear specification of what is required of an SCJVM or how it differs from a standard Java virtual machine. The specification of requirements needs to consider the requirements imposed, both explicitly and implicitly, on virtual machines by the SCJ specification [67] as that provides the authoritative source for information on SCJ. It is also helpful to consider the approach taken by some existing SCJVMs on points where the SCJ specification is unclear. The virtual machine must also meet the standard Java Virtual Machine specification [64] on points such as how to interpret Java bytecode instructions. There is much existing work on the semantics of Java bytecode that can be used in our work [16, 50, 112].

1.2.2 Compilation Strategy

As many existing SCJVMs precompile programs to native code in order to allow faster execution on embedded systems, it seems wise to include that in our approach. We focus on compilation of Java bytecode to C as that is the approach adopted by several existing virtual machines for embedded systems, including Fiji VM [93] and icecap HVM [110], and C is already widely used for embedded systems software.

There are two main approaches to specification and verification of compilers: the commuting diagram approach and the algebraic approach. The commuting diagram approach involves specifying the compiler as a function from the source language to the target language and showing that it is consistent with transformation between the semantics of the source and target languages [80, 114]. This approach has been used in much of the work on compiler correctness, including some of the earliest work [75] and more recent work such as that of the CompCert project [59, 60].

The algebraic approach involves defining the source and target languages in the same specification space, and using proved specialised rewrite rules to characterise compilation as model transformation in the extended language. This approach was first proposed in the early nineties by Hoare [44] and further developed by Sampaio [46, 101]. The algebraic approach does not seem to be as popular as the commuting diagram approach, but it does have the advantage that the specification of the compilation strategy is correct by construction as the rewrite rules that comprise it have all been proved.

We adopt the algebraic approach in our work, since it does not require the additional function that is required in the commuting-diagram approach because the source and target languages are defined in terms of the same specification language. The algebraic approach also permits a modular approach to proof and allows for the compiler to be easily implemented by application of the refinement rules using a term rewriting system. This means we can more easily evaluate the compilation strategy.

1.2.3 Formal Model and Proofs

As we are following the algebraic approach, we require a specification language in which to define the source and target languages. The use of a formal specification language for our specification of an SCJVM allows us to ensure that the specification is precise and to facilitate proofs of its correctness. This is beneficial for both the parts of the SCJVM that are involved in the compilation and the parts are not.

We have chosen *Circus* [89] as the specification language as it contains a wide variety of constructs that allow for specification of both data and behaviour, with an inbuilt notion of refinement, which we require for specifying the compilation strategy. *Circus* has also been used for previous work on the specification of SCJ programs [25, 26].

It is important that the correctness of the formal models and compilation strategy can be shown via mathematical proof, which requires the specification language to have a well-defined seman-

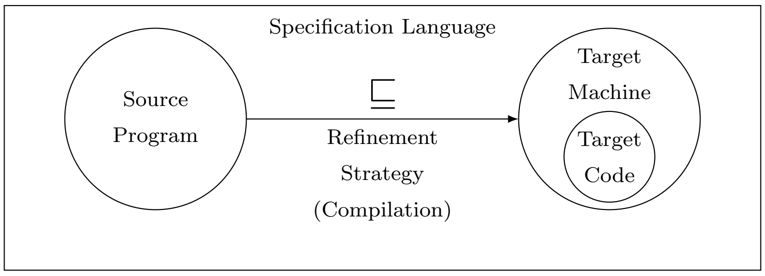


Figure 1.1: Standard algebraic approach

tics. *Circus* has such a semantics, defined using the model of Unifying Theories of Programming (UTP) [47].

1.2.4 Summary

In conclusion, our objective is an approach to verification of SCJVMs consisting of mechanised formal models together with proofs of properties about them. These formal models will cover both the services that must be provided by a running SCJVM and a compilation strategy for translating Java bytecode to native code. With our results, SCJVM developers will be able to create provably correct ahead-of-time compiling SCJVM implementations and check the correctness of those implementations.

1.3 Approach

As mentioned above, we follow the algebraic approach to verifying compilation, refining our model of Java bytecode to a representation of C code. The standard algebraic approach, developed by Hoare and Sampaio, follows the form shown in Figure 1.1. The source program is defined by a shallow embedding in the specification language and this is then refined to a model of the target machine in the specification language that contains the target code for the program.

The standard algebraic approach is normally applied to compile from a high-level language to a low-level language executable in a target machine. Here, we adapt the approach to deal with a low-level source language, Java bytecode. While Java bytecode has some high-level features, particularly its notion of objects, we view it as low-level since it is unstructured, with control flow managed using a program counter.

Our approach can be viewed as the usual approach applied in reverse, starting with an interpreter containing the bytecode source program, and proving that it is refined by an embedding of the C code, as shown in Figure 1.2. The core services of an SCJVM, such as scheduling and memory management, must be available for both the source and target codes. This may be viewed as specialising the interpreter to the behaviour of a specific bytecode program, so our approach is, in part, an approach to verifying a program specialiser [51].

For a low-level language, a deep embedding is the natural method for representing its semantics, since it is defined in terms of how it is processed by a (virtual) machine. For the C code we must choose whether to use a shallow embedding, representing C constructs by corresponding *Circus* constructs, or a deep embedding, creating a *Circus* model that interprets the C code.

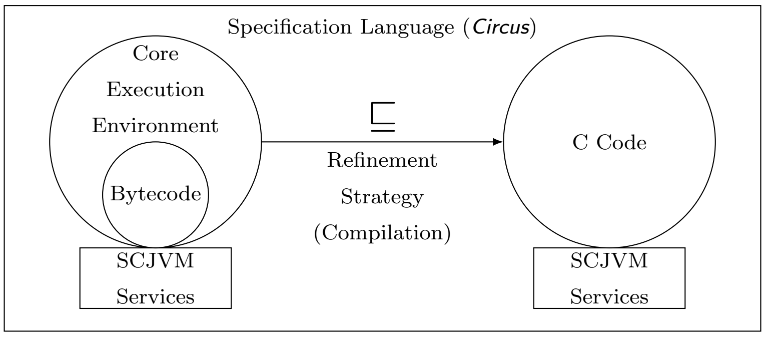

Specification Language (*Circus*)
Core Execution Environment
Bytecode
SCJVM Services
$\sqsubseteq$
Refinement Strategy (Compilation)
C Code
SCJVM Services

Figure 1.2: Our algebraic approach

We use a shallow embedding, since it allows existing algebraic laws for *Circus* to be used directly for manipulation of the C code and proof of the compilation rules. A deep embedding would require representing the syntax of C separately in *Circus* and rules for transforming the C code would have to be proved.

The shallow embedding approach is much easier to extend or adapt. If a larger subset of bytecodes needs to be considered or the target C code needs to be modified, in the worst case, we need more or different *Circus* compilation rules. There will be no need to extend the *Circus* model defining the C semantics.

As with previous applications of the algebraic approach, we divide our strategy into individual stages. However, since we are transforming from a low-level language to a high-level language, we eliminate a different part of the low-level machine state in each stage, rather than introducing it as in the previous applications of the approach.

In the first stage, the program counter is eliminated, and the control flow constructs of C are introduced to represent the control flow instead. The second stage then eliminates the Java frame stack from the state, introducing C variables to store the information that was stored on the stack. The third and final stage of the strategy then replaces the unstructured representation of memory used in the virtual machine with a representation of C structs.

Within each stage of the strategy, we specify algorithms for applying compilation rules. These compilation rules are algebraic laws, which we prove individually. Thus, we ensure each step of our compilation strategy is a correct transformation, as well as showing that the strategy as a whole performs the desired transformation.

1.4 Document Structure

Having given a brief overview of the area of study and identified the problem we wish to consider, the remainder of this dissertation proceeds as follows.

In Chapter 2 we examine the literature on safety-critical virtual machines and compilers for Java-like languages. This includes a discussion of why a safety-critical variant of Java is necessary and how it differs from standard Java. We also explain why a specialised virtual machine is necessary for SCJ. This is followed by a survey of the existing virtual machines for Safety-Critical Java and the techniques used in verifying compilers.

In Chapter 3 we present an identification of the requirements of SCJVM services, with a formal model of those requirements in the *Circus* specification language. This is followed by a model of an SCJVM core execution environment in Chapter 4, which includes the interpreter model that forms the starting point of our compilation strategy, and the C code embedding that is the target of our compilation strategy.

Then, in Chapter 5, we present our strategy for transforming SCJ bytecode executing in our interpreter model to our shallow embedding of C. This is divided into three stages, each of which is described in its own section, and explained using a running example. After this, in Chapter 6, we evaluate our model and compilation strategy, and discuss how they are validated as correct.

Finally, we conclude in Chapter 7 by summarising our contributions and mentioning the wider context of this research.

In addition to the chapters in the main body of this thesis, we also provide two Appendices, which contain information to support the understanding of the thesis. Appendix A contains all the compilation rules and laws used in the compilation strategy described in Chapter 5, providing a reference for compiler implementers to use. Appendix B contains the Java code of the examples discussed in Chapter 6 and their corresponding C code to aid in understanding the output of the compilation strategy and the discussion in Chapter 6.

We also provide an extended version of this thesis, with several additional appendices containing further information that may be of interest but which is not needed to understand the contents of this thesis. Appendix A of the extended thesis contains the full SCJVM services model described in Chapter 3. Similarly, Appendix B of the extended thesis contains the full CEE model described in Chapter 4. Appendix C of the extended thesis corresponds to Appendix A of this thesis, listing the rules and laws used in the compilation strategy. Appendix D of the extended thesis corresponds to Appendix B of this thesis, giving the code for the examples in Chapter 6. Appendix E of the extended thesis lists all the laws used in the proofs of the compilation rules from the strategy, including those laws that are not directly used in the strategy. Appendix F of the extended thesis provides theorems proved in Z/Eves as part of checking our models, with their corresponding proof scripts. Finally, Appendix G of the extended thesis contains hand-written proofs of the rules used in the compilation strategy.

Additionally, we note that the machine-readable sources for the Circus models, Z/Eves proofs and Java code produced in the course of writing this thesis are available online at `https://www.cs.york.ac.uk/circus/hijac/code/18-baxter_Thesis_Additional_Files.zip`.

Chapter 2

Compilers and Virtual Machines for Java-like languages in the Safety-critical Domain

This chapter begins with a discussion of why Java is being used in safety-critical systems and the need for a specialised version of Java for use in that area. Then, in Section 2.2, we discuss the variant of Java for real-time systems, and after that, in Section 2.3, we cover the variant of Java developed for safety-critical systems, discussing how they differ from standard Java and why a specialised virtual machine is required, before discussing some of the existing virtual machines for the safety-critical variant in Section 2.4.

In Section 2.5 we survey some of the literature on compiler correctness, and discuss the two main approaches in Sections 2.5.1 and 2.5.2, before seeing how the techniques of compiler correctness have been applied to Java-like languages in Section 2.5.3.

In Section 2.6, we give an overview of the *Circus* specification language used for our virtual machine specification, before concluding in Section 2.7.

2.1 Java for Safety-critical systems

In recent years Java has increasingly been considered as a language for writing safety-critical software. Other languages that are generally used in the safety-critical domain are C/C++ and Ada; C and C++ impose challenges concerning reliable use at the highest levels of safety [56], and the number of Ada programmers is not very large [17]. While Java has not traditionally been seen as a language for safety-critical systems, it was originally developed for the area of embedded systems, particularly for use in television set-top boxes, and has seen renewed interest in its use in embedded systems after gaining popularity in programming for the internet [83].

There are, however, several issues with standard Java that make it unsuitable for safety-critical systems. Many safety critical systems are also real-time systems, which are required to be predictable in their scheduling and use of memory. However, standard Java uses a garbage-collected memory model, which makes it hard to predict when memory may be freed or how long the process of freeing memory may take. Standard Java's thread model also lacks the predictability and control that is required in real-time systems.

To rectify these problems the Real-Time Specification for Java (RTSJ) [40] was created; it augments Java's memory and scheduling models with a system of scoped memory areas and a preemptive priority scheduler. RTSJ also allows for the standard Java models to be used alongside its own, making it suitable for a wide range of different real-time applications. On the other hand, this makes it hard to certify RTSJ applications and thus renders the RTSJ unsuitable for use in the safety-critical domain.

In order to allow certifiable safety-critical systems in Java, the Safety-Critical Java (SCJ) [67] specification was developed. SCJ is a subset of the RTSJ that leaves out the features from standard Java that are difficult to certify such as the garbage collector. SCJ also provides annotations that allow memory usage to be more easily checked. In the next section, we describe RTSJ in more detail, following which we discuss SCJ.

2.2 The Real-Time Specification for Java

RTSJ extends the scheduling and memory management models of Java with features that permit more predictable execution. In particular, RTSJ adds two types of schedulable objects to the threads of Java: real-time threads and asynchronous event handlers. These are scheduled by a real-time priority scheduling system in which each schedulable object has a priority and the highest priority object that is eligible to run at each point in time is the object that runs. This allows for simpler reasoning about order of execution and allows for more urgent tasks to preempt less urgent tasks.

The real-time threads of RTSJ run continuously from when they are started or repeatedly at regular intervals, unless they are interrupted by another schedulable object, or suspended waiting for a lock on an object. Asynchronous event handlers allow for code to execute in response to an event, which may be triggered by another schedulable object or by some external factor such as the hardware or operating system. Timers can also be used to trigger execution of an asynchronous event handler at specific time intervals.

RTSJ also provides mechanisms for preventing priority inversion, which is a situation in which a lower-priority schedulable object holding a lock on a resource required by a higher-priority schedulable object prevents the higher-priority schedulable object from executing, while the lower-priority schedulable object is itself blocked by other higher-priority schedulable objects. In particular, RTSJ supports priority ceiling emulation, in which a schedulable object is raised to the highest priority necessary to ensure it has priority over all threads that require the resource, and priority inheritance, in which the higher-priority schedulable object lifts the lower-priority schedulable object's priority when it requires the resource.

Since interruption by the garbage collector can make predicting execution time difficult, RTSJ also provides for real-time threads and event handlers that do not use the heap. Such schedulable objects cannot allocate on the heap or access objects allocated on the the heap, but also cannot be interrupted by the garbage collector. As alternatives to the heap for such schedulable objects, RTSJ provides immortal memory, in which allocated objects exist for the duration of the program, and scoped memory areas, which can be entered and exited as needed, with objects in the memory area being deallocated when the memory area is exited. These memory management methods allow for better predictability concerning when objects are deallocated and avoid the hard-to-predict interruptions associated with a garbage collector.

The additional features provided by RTSJ require support in the JVM used to execute an

RTSJ program, since the JVM must have an appropriate scheduler and must offer memory outside the heap. Several RTSJ virtual machines (RTSJVMs) have thus been created to run RTSJ programs. These include JamaicaVM [1], jRate [30], FijiVM [93], OVM [4] and Sun-Microsystems' Java for Real-Time Systems (Java RTS) [74]. Many RTSJVMs appear to be no longer maintained, including Sun's Java RTS. We also note that, since SCJ is based on RTSJ, some RTSJVMs also function as JVMs, and so we discuss them in more detail in Section 2.4.

Since many of the features offered by RTSJ are quite complex, and they are offered alongside standard Java alternatives such as (non-real-time) Java threads and the garbage-collected heap, RTSJ programs are hard to verify to the level required for safety-critical certifications. SCJ thus restricts the features of RTSJ to facilitate such certification. In the next section, we discuss SCJ in more detail, describing how SCJ programs are structured and how the features of RTSJ are restricted in SCJ.

2.3 Safety-Critical Java

SCJ removes the aspects of the RTSJ that make certification difficult, including standard Java threads and the garbage collector. This leaves scheduling and memory management models that are very different to the models for standard Java and that, therefore, require specialised virtual machines to support them.

SCJ defines three compliance levels to which programs and implementations may conform. Level 0 is the simplest compliance level. It is intended for programs following a cyclic executive approach. Level 1 lifts several of the restrictions of Level 0, allowing handlers that may trigger in response to external events and preempt one another. Level 2 is the most complex compliance level, allowing access to real-time threads and suspension via `wait()` and `notify()`.

An SCJ program consists of one or more missions, which are collections of schedulable objects that are scheduled by SCJ's priority scheduler. Missions are run in an order determined by a mission sequencer supplied by an SCJ program. Running a mission proceeds in several phases, as shown in Figure 2.1.

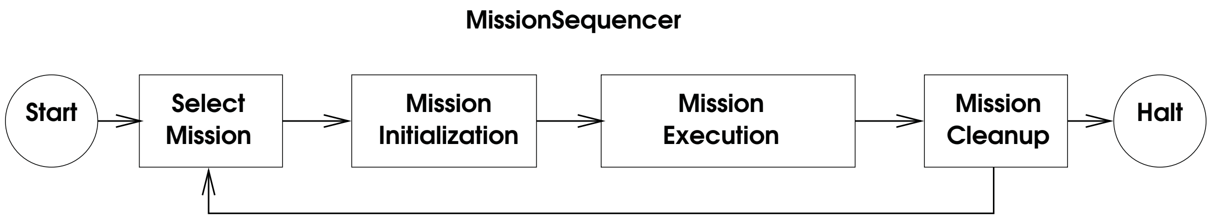


Figure 2.1: The phases of SCJ mission execution

The first phase is initialisation, which consists of setting up the schedulable objects controlled by the mission and creating any data structures required for the mission. Then the mission is executed by starting each of the schedulable objects in the mission and waiting for a request to terminate the mission. When the mission is requested to terminate, each of the schedulable objects in the mission is terminated and the mission's memory is cleared.

The schedulable objects within a mission are asynchronous event handlers that are released either periodically, at set intervals of time, aperiodically, in response to a release request, or once at a specific point in time (though handlers that are released once can have a new release time set, allowing them to be released again). At Level 2 real-time threads are also allowed.

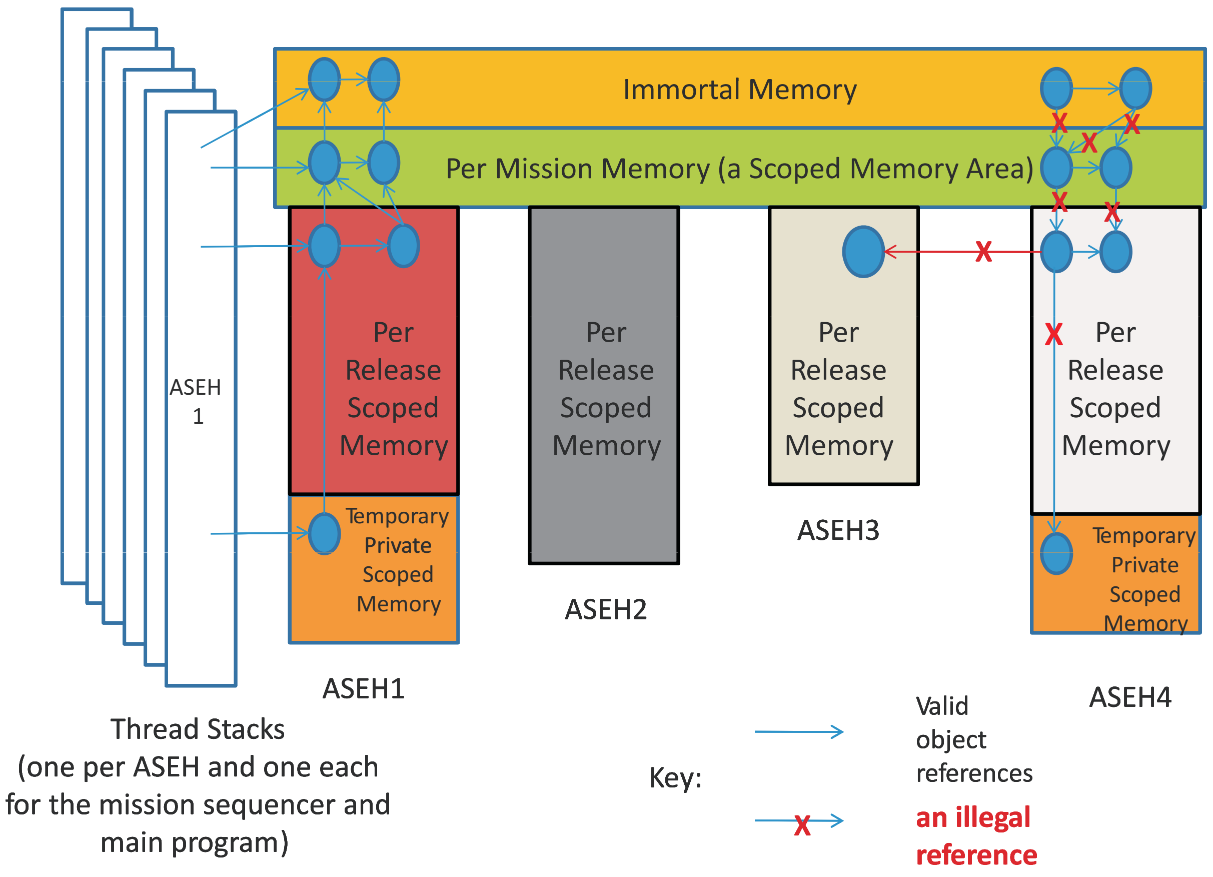

Immortal Memory
Per Mission Memory (a Scoped Memory Area)
ASEH 1
Per Release Scoped Memory
Temporary Private Scoped Memory
Per Release Scoped Memory
Per Release Scoped Memory
Per Release Scoped Memory
Temporary Private Scoped Memory
ASEH1
ASEH2
ASEH3
ASEH4
Thread Stacks (one per ASEH and one each for the mission sequencer and main program)
Key:
Valid object references
an illegal reference

Figure 2.2: An example of the layout of memory areas for four asynchronous event handlers (ASEHs), showing possible valid and invalid references between them

These schedulable objects are scheduled according to a priority scheduler as in RTSJ, but SCJ permits only priority ceiling emulation as a mechanism for avoiding priority inversion.

SCJ allows for assigning schedulable objects to “scheduling allocation domains”, where each domain consists of one or more processors. At Level 1, each scheduling allocation domain is restricted to a single processor. Hence, in scheduling terms, the system is fully partitioned. This allows for mature single processor schedulability analysis to be applied to each domain (although the calculation of the blocking times when accessing global synchronised methods are different than they would be on a single processor system due to the potential for remote blocking [31]).

SCJ deals with memory in terms of memory areas, which are Java objects that provide an interface to blocks of physical memory called backing stores. Memory allocations in SCJ are performed in the backing store of the memory area designated as the allocation context. Each schedulable object has a memory area associated with it that is used as the allocation context during a release of that object, and is cleared after each release. Each mission also has a mission memory area that can be used as an allocation context by the schedulable objects of that mission, to provide space for objects that need to persist for the duration of the mission or to be shared between the schedulable objects. The amount of memory required for the mission memory must be computed ahead of time and specified by the programmer as part of writing the mission, though there has been some work on automated computation of worst case memory use for SCJ programs [3]. There is also an immortal memory area where objects can be allocated if needed for the entire running of the program (they are never freed). SCJ places restrictions on which objects an object may point to, so as to avoid the creation of dangling pointers. Some examples of valid and invalid object references for some asynchronous event handlers are shown in Figure 2.2.

This system of memory areas in SCJ is based on the immortal memory and scoped memory of RTSJ, but it is fitted to the mission model of SCJ and explicitly excludes the possibility of allocating in a garbage-collected heap. This thus makes it easy to predict when memory is freed. However, it is not supported by standard JVMs as they do not provide memory outside of the heap for allocation and lack a notion of allocation context. The SCJ memory manager also needs to provide a means of accessing raw memory for the purposes of device access, which is mediated by accessor objects provided by the SCJ API. As this part of the API was stabilised at a later stage in the development of the SCJ specification than its other features, we have not had opportunity to include it in this work. It can, however, be seen that any system of raw memory access is not supported by most standard JVMs.

Moreover, dynamic class loading is not allowed in SCJ; all classes used by the program must be loaded when the program starts. This is because dynamic class loading may introduce time overheads that are hard to predict and additional code paths that complicate certification. Finally, SCJ also disallows object finalisers as it is not always easy to predict when they are run.

2.4 Virtual Machines for Safety-Critical Java

Because of the novel features of SCJ, briefly described in the previous section, a specialised virtual machine that provides support for allocation in memory areas and preemptive scheduling is required for SCJ. Although SCJ is a relatively recent development there have been various virtual machines created for SCJ or variations of SCJ, including JamaicaVM [1], icecap HVM [110], Fiji VM [93], OVM [4], HVM$_{\mathrm{TP}}$ [69], PERC Pico [6, 97] and JOP [98, 105]. These are each described in the following subsections.

2.4.1 JamaicaVM

JamaicaVM is an RTSJ virtual machine developed by aicas. Due to aicas' participation in the expert group developing SCJ and the fact that JamaicaVM is a very mature RTSJ implementation, it is used as the basis for the SCJ reference implementation, and is thus also an SCJVM. JamaicaVM supports both interpreting of Java bytecode and compilation to native code via C. When creating compiled code, JamaicaVM can perform profiling to improve the performance of the generated code. While JamaicaVM is a well-developed implementation of RTSJ and the reference implementation for SCJ, it is proprietary and so we cannot easily analyse its operation at the level of detail required for formal specification.

2.4.2 icecap HVM and HVM$_{\mathrm{TP}}$

The icecap hardware-near virtual machine (HVM) was created as part of the Certifiable Java for Embedded Systems Project [108] and provides an open-source implementation of SCJ targeted at embedded systems. The approach taken by the HVM is one of precompiling Java bytecode to C in order to allow for faster running programs with fewer memory resources. It includes an implementation of the SCJ libraries that covers most of SCJ level 2, originally supporting only single processor programs but with multiprocessor support added later [120]. This implementation, however, cannot be easily decoupled from the virtual machine itself.

The icecap HVM also provides a lightweight Java bytecode interpreter and allows for interpreted code to be mixed with compiled code. The reason for this is that the bytecode together with the interpreter can often be smaller than the compiled code, though there is a tradeoff for speed. HVM_{TP} is a modification of the icecap HVM's bytecode interpreter to improve time predictability and ensure that bytecode instructions are executed in constant time, which is important for ensuring real-time properties of the system hold.

2.4.3 Fiji VM

Fiji VM is a proprietary Java implementation designed to run on real-time embedded systems. Similarly to the icecap HVM, Fiji VM uses the strategy of compiling to C in order to improve performance. However, Fiji VM is not specifically targeted at SCJ and works with a range of libraries, including SCJ, RTSJ and the standard Java libraries. Fiji VM does have the advantage of high portability and multiprocessor support, which is lacking in some other SCJ virtual machines.

The fact that Fiji VM works with the SCJ libraries and supports the scoped memory model means it can run SCJ programs. It does not necessarily support all aspects of SCJ properly though.

2.4.4 OVM

OVM was created at Purdue University as part of the PCES project [9], to provide a virtual machine that can execute real-time Java programs with a high level of performance on embedded systems. Similar to Fiji VM and icecap HVM, OVM follows the principle of precompiling code for performance reasons, but translates Java to C++ instead of bytecode to C.

OVM also differs from the icecap HVM and Fiji VM in that it predates SCJ. It is written to implement the RTSJ, though it can still support SCJ programs; indeed, an SCJ implementation for OVM was later created [94]. However, OVM does not appear to have kept up with more recent changes to the draft SCJ standard. OVM is, unlike Fiji VM and the icecap HVM, single processor.

2.4.5 PERC Pico

PERC Pico is a product of Atego based on early ideas for SCJ, but uses its own system of Java metadata annotations to ensure the safety of scoped memory. This systems of annotations provides additional information about how memory is used so that it can be checked. Similarly to other SCJ virtual machines, PERC Pico allows for precompilation of Java code but targets executable machine code rather than an intermediate programming language. The metadata annotations are used to guide the compiler to produce code that uses the correct scoped memory. PERC Pico does not support the current SCJ standard, though it has been suggested that it could be modified to do so [84].

2.4.6 JOP

The Java Optimized Processor (JOP) [105] is a hardware-based implementation of a JVM, with time-predictability as a design goal. It allows for efficient execution of Java bytecode programs while also allowing analysis of real-time properties, particularly worst-case execution time. Because of this, it is well-suited to the applications that SCJ is aimed at and so an implementation of SCJ on JOP has been created [98]. This means that JOP forms an alternative SCJVM approach to that of ahead-of-time compilation. We focus instead on ahead-of-time compilation because, from the preceding discussion, it appears to be a more widely applied approach, since all of the 5 SCJVMs discussed above use ahead-of-time compilation to some language, with 3 of those compiling to C.

To summarise, as far as we are aware there is one publicly available ahead-of-time compiling virtual machine that has kept up with the developing SCJ specification, the icecap HVM. This is and, typically, virtual machines for SCJ will be, designed to be very small and fast so as to be able to run on embedded systems. As stated above, we focus on the common technique of running Java programs on embedded systems by precompiling them to native code. This means we must consider compiler correctness techniques to verify such a virtual machine; these techniques are discussed in the next section.

2.5 Compiler Correctness

Due to the importance of compiler correctness, there has been much research over the years in this area. Most of the work done follows a similar approach, which we term the commuting-diagram approach as it is based on showing that a particular diagram commutes. We discuss the commuting-diagram approach in Section 2.5.1.

An alternative approach to compiler verification is the algebraic approach developed in the early 90s. It is based on the concepts of refinement calculi designed for deriving software from specifications of behaviour. We explain the algebraic approach in Section 2.5.2 and discuss how it differs from the commuting-diagram approach.

We finish in Section 2.5.3 by reviewing some of the literature on correctness of compilers for Java-like languages. We explain how the techniques of compiler correctness have been applied in the case of Java and compare the different approaches.

2.5.1 Commuting-diagram Approach

Much of the work on compiler correctness can be seen as following the approach identified by Lockwood Morris [80], and later refined by Thatcher, Wagner and Wright [114]. The approach is essentially that a compiler correctness proof is a proof that the diagram shown in Figure 2.3 commutes, that is, $\gamma \circ \psi = \phi \circ \epsilon$.

Lockwood Morris had the corners of the diagram as algebras, rather than merely sets, with the functions between them being homomorphisms in order to add additional structure to the proof. This differs from the approach of some earlier works, particularly the earliest work by McCarthy and Painter [75], and instead follows work such as that of Burstall and Landin [20].

McCarthy and Painter's work featured a simple expression language with addition, natural numbers and variables. This was compiled to a simple 4-instruction single-register machine.

$$\begin{tikzcd}[column sep=large, row sep=large]
\text{source language} \arrow[r, "\text{compile}", "\gamma"'] \arrow[d, "\text{source semantics}"', "\phi"] & \text{target language} \arrow[d, "\psi"', "\text{target semantics}"] \\
\text{source meanings} \arrow[r, "\text{encode}", "\epsilon"'] & \text{target meanings}
\end{tikzcd}$$

Figure 2.3: The commuting diagram used in the traditional approach to compiler verification

The arrows of the diagram were simple functions, rather than homomorphisms, and the proof was performed using induction over the source language. This work laid the foundation for the study of compiler correctness.

Burstall and Landin showed correctness of a compiler for the same source and target languages as McCarthy and Painter; they used a more algebraic approach that better matches what Lockwood Morris later suggested. Burstall and Landin's approach involved representing the source and target languages, and their meanings, as algebras, with the compilation functions as homomorphisms. They targeted several intermediate machines in the proof of correctness. Viewing the languages as algebras allows for simpler proofs as some of the arrows of the commuting diagram can be wholly or partially derived from the algebraic structure. It was this goal of simplifying the proofs that led Lockwood Morris to advocate the use of algebras and homomorphisms.

The overall goal of pursuing formal proofs of compiler correctness, as proposed by McCarthy and Painter [75], is to allow machine-checked proofs of program correctness. There has been work in that area, the earliest of which was that by Milner and Weyhrauch [77] who showed the correctness of a compiler for an ALGOL-like language. The proof of correctness was partially mechanised in the LCF theorem prover [76] and the authors were of the opinion that the proof was feasible and could be completed relatively easily. A point to note is that Milner and Weyhrauch acknowledged the need for some way of structuring the proof in order to make it amenable to machine-checking. This gives further support to the algebraic commuting-diagram approach advocated by Lockwood Morris. Indeed, Milner and Weyhrauch explicitly followed that approach as they were in discussions with Lockwood Morris.

One advantage to making proofs easily machine-checkable, apart from the added certainty that the proof is correct, is that working compilers can be created from the machine-checked proofs. Code generation facilities are available with many theorem provers such as those of Isabelle/HOL [42] and Coq [62, 63]. The fact that the commuting-diagram approach involves treating the compilation as a function between algebras representing the source and target languages fits well with this idea. In this case, there is then a function defined in the mechanised logic for the purposes of conducting proofs about it that can be readily extracted to executable code.

The commuting-diagram approach has been followed in much of the literature through the years, though not always with the algebraic methods recommended by Lockwood Morris. The basic structure of the commuting diagram is a fairly natural approach to take, as seen by work such as that of the ProCoS project [21].

Another piece of work that follows the commuting-diagram approach is that of Polak [95], who states that he is more interested in verification of a “real” compiler rather than “abstract code generating algorithms”, and shows the correctness of a compiler for a Pascal-like language. This work focuses much more on pragmatic applications of the commuting-diagram approach, leaving behind the algebraic ideas of earlier papers. It sets a precedent for a simpler verification approach based on considering the functions in the commuting diagram.

The commuting-diagram approach has also been used in recent work, some of the most successful of which is that of CompCert [59–61]. This is a project to create a fully verified realistic compiler for a subset of C, using the theorem prover Coq [73].

There is also recent variation of the commuting-diagram approach, based on an operational semantics of the source language [8]. In this work, the operational semantics of the source language and a way of relating the source and target semantics are used to derive a different operational semantics of the source language acting on the state of the target machine. The semantics of the target language are then identified as part of that operational semantics and it is transformed to extract a compilation function. This approach may be viewed as variant of the commuting-diagram approach in which the compilation function is derived from the source and target semantics and the relationship between them, rather than being verified by those elements of the commuting diagram.

2.5.2 Algebraic Approach

The second main approach to showing correctness of compilers is the algebraic approach proposed by Hoare in 1991 [44], and further developed by Sampaio [46, 101, 102]. We note that the algebraic approach discussed in this section is largely unrelated to the algebraic commuting-diagram approaches mentioned in the previous section.

The algebraic approach to compilation derives from the concepts of algebraic reasoning about programs and program refinement. These concepts come from the idea, proposed by Hoare in 1984 [45], that programs can be thought of as predicates and so the laws of predicate logic can be used to construct laws for reasoning about programs [48]. As an example of such a law for reasoning about programs, we present below associativity of sequential composition, Equation (2.1), and left and right unit of sequential composition, namely, the program **Skip** that does nothing, Equation (2.2).

$$P;\ (Q;\ R) = (P;\ Q);\ R \tag{2.1}$$

$$P;\ \mathbf{Skip} = \mathbf{Skip};\ P = P \tag{2.2}$$

The notion of refinement is central to the algebraic approach to compilation. Refinement calculi have been developed, independently, by Back [7], Morris [81] and Morgan [79], following from earlier concepts of program transformation [5, 10, 11, 111]. The basic idea is that there is a relation between programs that captures the idea of one program being “at least as good as” another or, to put it more precisely, at least as deterministic as another. Languages and laws for reasoning about programs with this notion of refinement can then be used to develop programs from specifications. This means that certain aspects of a system can have a nondeterministic specification and several different implementations can refine that specification.

In using refinement to show the correctness of a compiler, the laws of the specification language can be used to prove compilation refinement laws. These compilation laws can be used to transform the source programs into some normal form that represents an interpreter for the target language running the target code. In other words, the code output by the compiler, when executed by on the target machine, must be a refinement of the source program. The compilation laws can be used to prove this refinement and at the same time generate the target code.

As an example, consider the following refinement in which a simple program that performs some arithmetic and stores the results into variables is refined by a normal form representing the target machine and code. The symbol $\sqsubseteq$ represents the refinement relation here.

$$\mathbf{var}\, x, y, z \bullet x := (x+5) \times (y+z);\; z := z+1 \sqsubseteq \begin{array}{l} \mathbf{var}\, A, P, M \bullet P := 1;\; \mathbf{do} \\ \quad P = 1 \longrightarrow A, P := M[2], 2 \\ \quad \Box\, P = 2 \longrightarrow A, P := A + M[3], 3 \\ \quad \Box\, P = 3 \longrightarrow M[4], P := A, 4 \\ \quad \Box\, P = 4 \longrightarrow A, P := M[1], 5 \\ \quad \Box\, P = 5 \longrightarrow A, P := A + 5, 6 \\ \quad \Box\, P = 6 \longrightarrow A, P := A \times M[4], 7 \\ \quad \Box\, P = 7 \longrightarrow M[1], P := A, 8 \\ \quad \Box\, P = 8 \longrightarrow A, P := M[3], 9 \\ \quad \Box\, P = 9 \longrightarrow A, P := A + 1, 10 \\ \quad \Box\, P = 10 \longrightarrow M[3], P := A, 11 \\ \mathbf{od};\; \{P = 11\} \end{array} \tag{2.3}$$

The normal form represents the behaviour of an interpreter for the target code running in a target machine whose structure is defined by the variables A, P, and M. The variable A represents a general-purpose register of the target machine, P represents the program counter of the target machine, and M is an array representing the memory of the target machine. The normal form consists of a program that initialises P to 1 and then enters a loop in which the operation performed on each iteration is dependent on the value of P. The loop is exited when P is set to a value for which there is no operation and it is asserted that P will be equal to 11 at the end of the program. Each of the statements of the source program corresponds to several operations in the normal form as complex expressions are broken down into simpler expressions that can be handled by instructions of the target machine.

The compilation proceeds by first applying rules to simplify the assignment statements. The register A is introduced at this stage by splitting assignments of expressions to variables into two assignments that transfer the values to and from A. In this way, the assignments are transformed for the target machine that only has instructions involving registers. Particularly complex expressions such as $(x + 5) \times (y + z)$ are handled by storing intermediate results in temporary variables. In this case the result of the expression $y + z$ is placed in a temporary variable when $P = 3$. The variables used in the source program and introduced compilation are later replaced with locations in the memory array M in a data refinement step. This causes the variables x, y and z to be replaced with $M[1]$, $M[2]$ and $M[3]$ respectively. The temporary variable introduced to store the result of $y + z$ is similarly replaced with $M[4]$.

Each of the assignment statements is then refined by a normal form with an explicit program counter P, that is incremented as part of the assignment operation. These normal forms are then combined together by the refinement rule for sequential composition to create the normal form of the full program. The update of the program counter in this program is quite simple but more complex updates would occur for conditionals or loops.

The power of the algebraic approach is that the compilation of individual elements of the source language can be specified and proved separately in different refinement laws. The compilation can also be split into stages, with a set of refinement laws for each stage to modularise the compilation. The separate refinement laws can then be combined to form a compilation strategy.

The first major work done using the algebraic approach was that of Sampaio [101], who used it to specify a correct compiler for a simple language that, nonetheless, covers all the constructs available in most programming languages. The target machine Sampaio used was a simple single-register machine that bears similarity to most real processor architectures. He mechanised the compiler in the OBJ3 term rewriting system [39], showing that working compilers can be easily created from specifications using the algebraic approach. However, the algebraic laws Sampaio used to prove correctness of the compiler were taken as axioms. Sampaio notes that they could be easily proved given a semantics for the reasoning language.

Though there has not been much work done using the algebraic approach, we single out the work of Perna [91, 92], showing correctness of a compiler for a hardware description language. The compilation takes high-level descriptions of hardware written in Handel-C and transforms them into systems of basic hardware components connected by wires. The algebraic approach works well here as the target language is a subset of the source language, albeit in a different form. Perna was able to handle features not covered by most other works on hardware compilation, such as parallelism with shared variables. Also, whereas Sampaio took the basic algebraic laws as axioms, Perna proved the laws from a semantics given using the Unifying Theories of Programming (UTP) model [47]. There has also been work on the correctness of Java compilers using the algebraic approach. This is considered in the next section, where we consider compiler correctness for Java-like languages.

2.5.3 Correctness of Java Compilers

The popularity of Java has meant that there has been plenty of work on formalising Java and the JVM [43], but there have been relatively few works on formally verified compilers for Java-like languages. However, the work that has been done uses both of the two main approaches and covers most of the features of Java.

Some of the earliest and most thorough work is that by Särk, Schmid and Börger [112], who formalise most of Java and the JVM before specifying and showing the correctness of a compiler for Java. The approach taken by them uses Abstract State Machines (ASMs) to specify the source and target languages. The ASMs give an operational semantics to Java and the JVM, describing how each construct affects the running of the program. The languages are each specified by multiple ASMs, beginning with an imperative core, then adding classes, objects, exceptions and, finally, threads.

Although this approach is called the ASM approach, it becomes clear from the definition of compiler correctness given in terms of a mapping between ASMs that this work ultimately follows the commuting-diagram approach. This work leaves parts of the proof incomplete (in particular, compilation of threads is not addressed) and applies to an old version of Java. This is, nevertheless, an admirable attempt at producing a verified Java compiler.

Work has also been done by Duran following the algebraic approach [34, 35]. Duran's work specifies a compiler for a language called Refinement Object-Oriented Language (ROOL) [19], which was created for reasoning about object-oriented languages and bears much similarity to Java. ROOL features constructs for specifying and reasoning about programs as well as object-oriented programming language constructs. This means that the there are algebraic laws for ROOL, from which the rewrite rules that form the basis of the algebraic approach can be proved. Duran's work adds further phases to Sampaio's compilation strategy in order to deal with the object-oriented features, but does not consider some other aspects of Java such as exceptions and threads. Duran notes that other work has addressed some of those issues.

While the two works already discussed were not machine checked, there have also been compiler correctness proofs for Java-like languages in the Isabelle/HOL proof assistant. The first of these was by Strecker [113], showing correctness of a compiler for a subset of Java called μJava, which already had a formalisation of its semantics in Isabelle/HOL [85]. This work was followed by Klein and Nipkow's work on a compiler for a slightly larger subset of Java called Jinja [54], which added exception handling. Finally, Lochbihler [65] added threads to Jinja and showed correctness of compilation for Java concurrency. It is notable that this is the only work on Java compilation that properly addresses concurrency. All of these works follow the commuting-diagram approach.

Though some work has been done on correct compilers for Java-like languages and many virtual machines for SCJ adopt an approach of compiling to native code, no work has been done on verifying that compilation to native code. Therefore, in this thesis, we consider correctness of the compilation to native code as part of our work on SCJ virtual machines. We follow the algebraic approach as it gives greater assurance of correctness, as an additional function mapping source meanings to target meanings is not required, and a good level of modularity, as the compilation is split into separately proved rewrite rules. In order to represent the normal form we require a specification language and for that purpose use *Circus*, which is described in the next section.

2.6 *Circus*

The *Circus* specification language [89] is based on CSP [99], which is used to specify processes that communicate over channels, and the Z notation [118], which is used to specify state and data operations. A *Circus* specification is made up of processes that communicate over channels. These channels may carry values of a particular type, or may be used as flags for synchronisation or signalling between processes. Each process may have state, and is made up of actions that operate on that state and communicate over channels.

Circus is a language for refinement. It allows for a program's behaviour to be written as an abstract specification, including invariants and nondeterminism. After reasoning using the invariants present in the abstract model to ensure it yields the desired behaviour, a refinement can be established to a more concrete model from which an executable program can be created. The provision for refinement in *Circus* makes it well suited for use as a specification language for the algebraic approach to compilation.

We illustrate the concepts of *Circus* using as an example the process for the real-time clock from an early version of our specification of an SCJ virtual machine. The specification begins with a declaration of the channels that may be used in the following processes. Type declarations written in Z can also be included at the beginning of a *Circus* specification. Here, we define a type *Time* to be the set of natural numbers and create a boolean datatype

$$\begin{aligned}&\mathit{Time} == \mathbb{N}\\&\mathit{Bool} ::= \mathit{True} \mid \mathit{False}\end{aligned}$$

We declare channels to represent interactions corresponding to calls to methods to get the clock's time and precision, and set and clear alarms. Channels are also declared to model interactions with the hardware that accept clock tick interrupts and read the time from the hardware clock.

$$\begin{aligned}&\mathbf{channel}\ \mathit{getTime}, \mathit{getPrecision}, \mathit{setAlarm} : \mathit{Time}\\&\mathbf{channel}\ \mathit{clearAlarm}\\&\mathbf{channel}\ \mathit{HWtick}\\&\mathbf{channel}\ \mathit{HWtime} : \mathit{Time}\end{aligned}$$

We also specify a constant to represent the clock's precision using a Z axiomatic definition. The value of the constant is required to be nonzero, but is otherwise left unrestricted, so that any nonzero time value is a valid instantiation.

$$\begin{array}{|l}\mathit{precision} : \mathit{Time}\\\hline\mathit{precision} > 0\end{array}$$

After the channel declarations, we can declare processes that use them. Here we declare the *RealtimeClock* process. It is a basic process, that is, its state is defined in Z, and its behaviour using CSP constructs and Z data operations.

$$\mathbf{process}\ \mathit{RealtimeClock} \mathrel{\widehat=} \mathbf{begin}$$

In this example, the state records the current time, whether an alarm is set, and the time of the alarm that may be set. An invariant specifies that if an alarm is set, then the time of the alarm must not be in the past.

$$\begin{array}{|l}\hline \mathit{RTCState}\\\mathit{currentTime} : \mathit{Time}\\\mathit{alarmSet} : \mathit{Bool}\\\mathit{currentAlarm} : \mathit{Time}\\\hline\mathit{alarmSet} = \mathit{True} \Rightarrow\\\quad \mathit{currentAlarm} \geq \mathit{currentTime}\\\hline\end{array}$$

$$\mathbf{state}\ \mathit{RTCState}$$

The behaviour is described using actions, written in a mixture of Z and CSP. The first action is a Z initialisation operation, *Init*0. Its final state is represented by variables obtained by placing a prime on the names of the state components. Here, the initialisation takes as input the initial time, represented by the variable *initTime*?. In Z schemas, inputs to operations are distinguished by ending with a question mark. Similarly, outputs are marked with an exclamation mark. The current time is defined to be equal to the initial time and no alarm is initially set. The initial time of the alarm is arbitrary, that is, nondeterministically chosen from elements of its type, since the initialisation imposes no restrictions on it.

$$\begin{array}{|l}
\hline \mathit{Init0} \\
\mathit{RTCState}' \\
\mathit{initTime}? : \mathit{Time} \\
\hline
\mathit{currentTime}' = \mathit{initTime}? \\
\mathit{alarmSet}' = \mathit{False} \\
\hline
\end{array}$$

The action *Init*, defined below, uses a CSP prefixing to specify an input communication before the initialisation operation *Init*0. The initial time of the clock is read from the hardware clock and then the initialisation specified by the Z schema is performed.

$$\mathit{Init} \mathrel{\widehat=} \mathit{HWtime}?\mathit{initTime} \longrightarrow \big(\mathit{Init0}\big)$$

The action that returns the current time simply uses CSP to output the current time from the state over the *getTime* channel. The action ends with the special action **Skip**, which indicates the end of an action.

$$\mathit{GetTime} \mathrel{\widehat=} \mathit{getTime}!\mathit{currentTime} \longrightarrow \mathbf{Skip}$$

Setting a new alarm is a more complex operation that involves Z schemas that specify two different scenarios in which this operation may be used. In the first case, the new alarm is not in the past. The symbol Δ denotes a change of state. The operation stores the time of the new alarm and sets a flag to indicate an alarm is set in this case.

$$\begin{array}{|l}
\hline \mathit{SetAlarm0} \\
\Delta \mathit{RTCState} \\
\mathit{newAlarm}? : \mathit{Time} \\
\hline
\mathit{newAlarm}? \geq \mathit{currentTime} \\
\mathit{currentAlarm}' = \mathit{newAlarm}? \\
\mathit{alarmSet}' = \mathit{True} \\
\mathit{currentTime}' = \mathit{currentTime} \\
\hline
\end{array}$$

In the second case, the new alarm is in the past and so the alarm is not set (we have omitted the error reporting for the sake of simplicity). The symbol Ξ denotes that the state remains the same.

$$\begin{array}{|l}
\hline \mathit{SetAlarm1} \\
\Xi \mathit{RTCState} \\
\mathit{newAlarm}? : \mathit{Time} \\
\hline
\mathit{newAlarm}? < \mathit{currentTime} \\
\hline
\end{array}$$

The two Z schemas are combined using a logical disjunction, allowing either to specify the behaviour when a request to set the alarm takes place.

$$\mathit{SetAlarm} \mathrel{\widehat=} \mathit{setAlarm}?\mathit{newAlarm} \longrightarrow \big(\mathit{SetAlarm0} \lor \mathit{SetAlarm1}\big)$$

In addition to Z and CSP constructs, *Circus* also has other constructs more familiar to programmers, such as if statements and do loops. One of these constructs, the assignment operator, is used in the action that clears the current alarm to update part of the state without requiring a Z schema. The alarm is cleared by simply setting *alarmSet* to *False*, without updating any other state variables.

$$ClearAlarm \mathrel{\widehat=} clearAlarm \longrightarrow alarmSet := False$$

Each of the actions the process can perform are joined together with the CSP external choice operator, which chooses an action to take based on the channel communications that the environment is willing to perform. This includes the actions above, as well as some other actions that have been omitted here. The choice is repeated in a loop.

$$\begin{aligned}&Loop \mathrel{\widehat=} (GetTime \mathbin{\Box} SetAlarm \mathbin{\Box} ClearAlarm \mathbin{\Box} \cdots)\\&\qquad ;\ Loop\end{aligned}$$

The *Circus* process then ends with the main action that specifies the overall behaviour of the process. Here, the process simply performs the initialisation and then enters the loop.

$$\bullet\ Init\ ;\ Loop$$

end

In addition to the constructs presented here *Circus* also contains operators for composing processes in parallel, with or without synchronisation on channels. These operators are used both to specify actual parallelism and to represent composition of requirements. In this way several *Circus* specifications of individual components can be combined to form a specification of the entire system.

A detailed account of *Circus* can be found in [89]. Table 2.1 summarises the *Circus* constructs used in this thesis.

2.7 Final Considerations

We have seen that Java is increasingly being considered as a language for safety-critical embedded systems and that the modifications to Java required to make it suitable for such systems require a specialised virtual machine. The developing Safety-Critical Java specification has several differences from standard Java, particularly in the areas of scheduling and memory management, that make standard JVMs unsuitable for running SCJ programs. We have considered several virtual machines that have been developed for running SCJ programs and noted that none of them has been formally verified and that most of them adopt an approach of precompiling programs to native code.

With that in mind, we have considered the techniques used to verify the correctness of compilers and found that there are two main approaches: the commuting-diagram approach and the algebraic approach. In the commuting-diagram approach the source semantics, target semantics, compilation function, and a function mapping the source meanings to the target meanings, are shown to commute. This approach is popular and has had much research done on it but relies on the definition of the function from the source meanings to the target meanings.

The algebraic approach defines the source and target languages within the same specification language, which is additionally equipped with a refinement relation between programs. Laws of the specification language are then used to prove refinement rules that are applied according to

Construct	*Circus* notation
Termination	**Skip**
Divergence (abortion)	**Chaos**
Assignment of expression e to variable x	$x := e$
Prefixing of signal on channel c to action A	$c \longrightarrow A$
Prefixing of output on channel c of expression e to action A	$c!e \longrightarrow A$
Prefixing of input on channel c of variable x to action A	$c?x \longrightarrow A$
Variable block with variable x, of type T, and action A	$\mathbf{var}\, x : T \bullet A$
Value parameter block with parameter x, of type T, and action A	$\mathbf{val}\, x : T \bullet A$
Result parameter block with parameter x, of type T, and action A	$\mathbf{res}\, x : T \bullet A$
Instantiation of parameterised action A with expression e	$A(e)$
Guarding of A with predicate g	$(g) \mathbin{\&} A$
Sequential composition of actions A and B	$A \mathbin{;} B$
External choice of actions A and B	$A \Box B$
Conditional choice of actions A and B, with conditions g and h	$\mathbf{if}\ g \longrightarrow A \mathbin{[\!]} h \longrightarrow B\ \mathbf{fi}$
Parallel interleaving of processes P and Q	$P \mathbin{\vert\vert\vert} Q$
Recursion with body given by action function F	$\mu X \bullet F(X)$
Parallel composition of processes P and Q, synchronising on the intersection of channel sets $cs1$ and $cs2$	$P_{cs1}\Vert_{cs2}\ Q$
Parallel composition of processes P and Q, synchronising on the channel set cs	$P \llbracket\ cs\ \rrbracket\ Q$
Hiding of channel set cs in process P	$P \setminus cs$

Table 2.1: Summary of *Circus* notation

some compilation strategy. The algebraic approach has the advantage that it does not require the additional function that is required in the commuting-diagram approach, since the source and target languages are defined in terms of the same specification language. The algebraic approach also permits a modular approach to proof and allows for the compiler to be easily implemented by application of the refinement rules using a term rewriting system.

Given the considerations above, we have decided to adopt the algebraic approach when specifying the compilation to native code employed by many SCJ virtual machines. This means that a specification language is required in which to define the source and target languages, as well as for the purposes of specifying other aspects of the virtual machine. We have chosen *Circus* as the specification language as it contains a wide variety of constructs that allow for specification of both data and behaviour, has a well defined semantics with many laws already proved, and has been used for previous work on the specification of SCJ programs. *Circus* also has some existing mechanisation and tool support, which can help give greater assurance of the correctness of specifications.

We note that our work in this thesis particularly focusses on SCJ Level 1 programs executing on a single-processor SCJVM. Consideration of multiprocessor SCJVMs and Level 2 programs is left to future work. This follows the approach of other works in this area, which have focused on the core features of SCJ in initial work. An example of this approach is in the development of SCJ programs from specifications, with Level 1 programs considered in [25, 26] and the work later extended to Level 2 programs in [68]. Similarly, we recall that icecap initially only supported single-processor programs, with multiprocessor support added to it later.

Chapter 3

Safety-Critical Java Virtual Machine Services

In order to reason about a Safety-Critical Java virtual machine (SCJVM), we first require an identification of the requirements of an SCJVM and a formal model of those requirements. For the purposes of our model, we consider an SCJVM to have the components illustrated in Figure 3.1. An SCJVM is divided into two main parts: the core execution environment, and the SCJVM services, which may make use of the services of an underlying operating system or hardware abstraction layer.

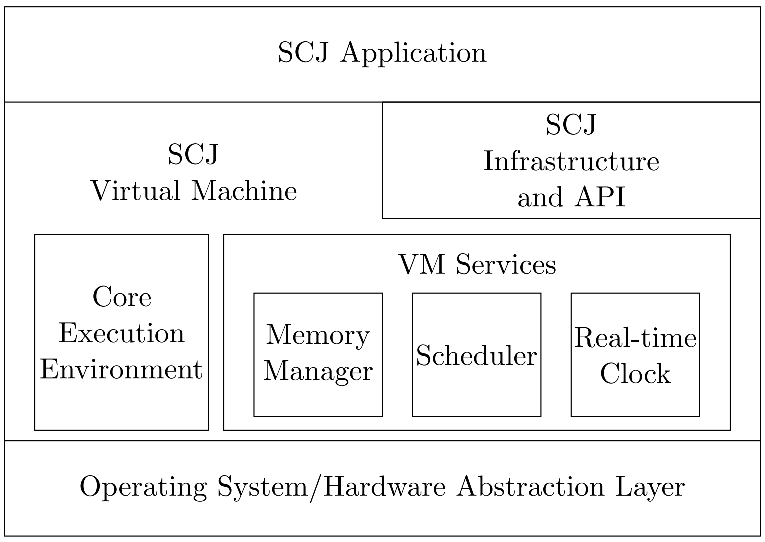

SCJ Application
SCJ Virtual Machine
SCJ Infrastructure and API
Core Execution Environment
VM Services
Memory Manager
Scheduler
Real-time Clock
Operating System/Hardware Abstraction Layer

Figure 3.1: A diagram showing the structure of an SCJVM and its relation to the SCJ infrastructure and the operating system/hardware abstraction layer, focusing on the SCJVM services

The core execution environment manages the execution of Java bytecode, whether that be via interpretation, just-in-time compilation or ahead-of-time compilation. The core execution environment must also manage data that relates to the execution of bytecode instructions, such as the representation of classes and objects.

The SCJVM services represent the additional services that must be offered by an SCJVM in order to support the SCJ infrastructure. These services may be supplied as standalone services and so do not need to be handled by the compilation strategy. We consider the virtual machine services to be divided into three areas:

- the memory manager, which manages backing stores for memory areas and allocation within them;
- the scheduler, which manages threads and interrupts, and allows for implementation of SCJ event handlers; and
- the real-time clock, which provides an interface to the system real-time clock.

Each of these services is used either by the core execution environment or by the SCJ infrastructure. Some of the services also rely on each other. For example, the real-time clock must communicate with the scheduler to trigger an interrupt handler when an alarm's time passes.

A model of the core execution environment is presented in Chapter 4. In this chapter, we present the requirements for each area of the SCJVM services: the memory manager in Section 3.1, the scheduler in Section 3.2, and the real-time clock in Section 3.3. The formal model of the SCJVM requirements is presented in Section 3.4. A complete version of the model can be found in Appendix A of the extended version of this thesis [13].

The memory manager model has been subject to proof using Z/Eves. The theorems proved about the memory manager, and their Z/Eves proof scripts can be found in Appendix F of the extended version of this thesis [13].

Part of an earlier version of this model was presented at the 13th International Workshop on Java Technologies for Real-time and Embedded Systems [14] with the full earlier version made available as a technical report [12].

3.1 Memory Manager API

The SCJVM memory manager deals with the raw blocks of memory used as backing stores for the memory areas of SCJ. The memory areas themselves are Java objects, and so are dealt with by the core execution environment and accessed through the SCJ API, instead of directly via the virtual machine. This is in line with what is specified in the SCJ standard and also done for RTSJ. Backing stores are assumed to have unique identifiers that can be used to refer to them; these identifiers can be simply pointers to the physical blocks of memory used for backing stores.

Each backing store is composed of two parts: an area of memory in which memory for objects may be allocated, and an area in which other backing stores can be allocated. A backing store may thus have other backing stores nested within it, so that a possible memory layout is as shown in Figure 3.2. There, we divide the two parts of each backing store by a dashed line, with the object allocation area below and the backing store area above.

There is initially one backing store, called the root backing store, which has its size set when the SCJVM starts up to cover all the memory available for allocation in backing stores. The root backing store cannot be destroyed, so that there is always a fixed base for the layout of memory. The root backing store is used as the backing store for the immortal memory area. The root backing store initially has all its space available for object allocations, with no space to allocate nested backing stores. The infrastructure must reduce the object allocation space to match the space required by the `Safelet` during SCJVM startup.

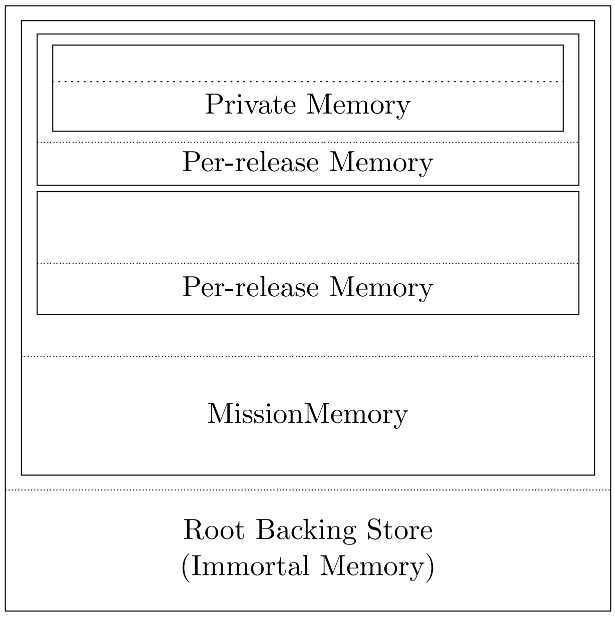


Figure 3.2: An example memory layout

The operations of the memory manager API are summarised in Table 3.1. In addition to the inputs and outputs described there, there should also be some system of reporting erroneous inputs, whether that be exceptions, global error flags, or particular return values signalling errors. The conditions that cause an error to be reported are listed in Table 3.1 as well.

The memory manager operations presented here are intended to support implementations of the SCJ memory API, rather than directly implement the API itself. Implementation of the SCJ API operations is part of the core execution environment (CEE), and so is described in Chapter 4. For example, the backing store operations presented here take a backing store as input, since tracking of the current memory area (with its underlying backing store) is handled by the CEE. So memory area entering operations are also implemented in the CEE. Similarly, the memory manager presented here allows for allocating raw blocks of memory rather than allocating objects, since the structure of objects is defined by the CEE. Hence the operation of creating a new object is defined there using the memory-allocation operation presented here.

The root backing store is always available to the SCJ infrastructure through the `getRoot-BackingStore` operation. An SCJ program, on the other hand, does not have direct access to the root backing store except through memory areas provided by the infrastructure.

It is possible to obtain information about the used and available space in the object allocation area of a given backing store using the operations `getTotalSize`, `getUsedSize`, and `getFree-Size`. This information is made available to SCJ programs through the interface provided by memory areas defined in the infrastructure. Similarly, the `getRemainingBackingStore` operation provides the amount of free space in the area for allocating nested backing stores.

The backing store in which a particular memory address lies can also be queried. This information can be obtained by the `findBackingStore` operation and is required by the infrastructure for obtaining the memory area of a given object. This operation fails if the address is not the address of an object, since this is intended for determining the backing store of an object pointer, not other addresses in a backing store.

Operation	Inputs	Outputs	Error Conditions
`getRootBackingStore`	(none)	backing store identifier	(none)
`getTotalSize`	backing store identifier	size in bytes	invalid identifier
`getUsedSize`	backing store identifier	size in bytes	invalid identifier
`getFreeSize`	backing store identifier	size in bytes	invalid identifier
`getRemainingBackingStore`	backing store identifier	size in bytes	invalid identifier
`findBackingStore`	memory pointer	backing store identifier	not in object space
`allocateMemory`	backing store identifier size in bytes	memory pointer	invalid identifier insufficient free memory
`makeBackingStore`	backing store identifier total size in bytes allocation area size in bytes	backing store identifier	invalid identifier insufficient free memory
`clearBackingStore`	backing store identifier	(none)	invalid identifier
`resizeBackingStore`	backing store identifier size in bytes	backing store identifier	invalid identifier backing store not empty new size too small new size too large
`createStack`	size in bytes	stack identifier	insufficient free space
`destroyStack`	stack identifier	(none)	invalid identifier stack space fragmentation

Table 3.1: The operations of the SCJVM memory manager

Allocation within backing stores is possible through the `allocateMemory` operation, which allocates blocks of memory within a given backing store. This operation is provided for the core execution environment to implement the `new` bytecode instruction and is not directly available to the program or infrastructure. Though the memory manager allocates space for objects, there is no notion of objects in the memory manager since they only exist at the level of Java code, and so are dealt with by the core execution environment. Dealing solely with blocks of memory in the SCJVM services allows objects to be represented in a way appropriate to the structure of the core execution environment. Allocations within backing stores must not cause fragmentation, so as to fulfil real-time predictability requirements. The operation `allocate-Memory` must also zero the memory it allocates, in order to match the semantics of `new`.

Allocation of backing stores is provided by `makeBackingStore`, which is available to the infrastructure for use when creating new memory areas. A new backing store is created nested within the specified backing store. The total size of the new backing store (including space for allocating nested backing stores) must be specified when using this operation, along with the space required for object allocations. The infrastructure is responsible for storing the backing store identifier returned by `makeBackingStore`. Backing store allocation must be done in constant time without fragmentation.

Deallocation of memory in backing stores cannot be done directly as that could introduce fragmentation and would defeat the scoped-memory model of SCJ. Instead, the SCJVM provides for clearing a backing store when the memory area it serves is no longer in use. This functionality is provided by the operation `clearBackingStore`, which clears the specified backing store, deallocating all objects and nested backing stores within it. It is not necessary to track exactly which objects are deallocated by this operation as SCJ does not have object finalisers. The clearing of a backing store includes the clearing of all backing stores nested within it, whose memories are freed with the rest of the backing store. This would create a problem if the parent backing store were cleared while another thread is using a backing store within it as an allocation context. Such a situation should not occur as the backing stores of mission memory and immortal memory are the only ones that contain backing stores in use by different threads.

The mission memory is only cleared when all the event handler threads within the mission have finished and the immortal memory should never be cleared.

The last operation on backing stores is their resizing. This is provided for by the operation `resizeBackingStore`, which resizes the object allocation area of a backing store by moving the boundary between the object allocation area and the nested backing store area. To ensure that this does not fragment existing used memory in the backing store, it is required that either the backing store is empty (i.e. it contains no objects or backing stores), or it is entirely composed of space for object allocations. This is acceptable, since this operation is only needed for resizing of the mission memory inbetween missions, resizing of a nested private memory when it is reentered, and resizing the immortal memory during SCJVM startup. For resizing mission memory inbetween missions, it should be resized up to the maximum available size after the mission has finished (during which it has been cleared), and then resized down to the required size after the mission object has been obtained (during which it only contains object space as it covers the whole space available). Private memory areas are always cleared before being resized. The root backing store (used for immortal memory) is entirely composed of space for object allocations when the SCJVM starts, allowing this operation to be used in that case also. In the case where the backing store is not empty, the new size must be sufficient to contain any existing object allocations.

These operations on backing stores each take a backing store identifier as input since the memory manager does not handle allocation contexts. Management of allocation contexts is instead left to the core execution environment, which must pass the appropriate backing store identifier when using the memory manager services.

The memory manager must also manage stacks, which are placed in a separate area of memory to the backing stores. The operations `createStack` and `destroyStack` allow for stacks to be created and destroyed. The stack space must not be fragmented, which is a requirement that can be met since stacks for threads are allocated together when a mission is initialised and destroyed together when the mission ends. That remains true at level 2 where nested missions are permitted, since the nested mission's stacks are allocated after the stacks of its parent mission, and are destroyed before the parent mission ends. Like backing stores, stacks are referred to by unique identifiers that may simply be pointers to the space allocated for the stack.

In the next section we give an overview of the second area of SCJVM services, the scheduler.

3.2 Scheduler API

The SCJVM scheduler manages the scheduling of threads, which are abstract lines of execution, each with its own stack and current allocation context. These threads are useful, for example, to implement the event handlers of SCJ, with each event handler being bound to a single thread. The operations of the scheduler are summarised in Table 3.2.

Each thread is scheduled according to a priority level. The SCJ standard requires that there be at least 28 priorities and separates them into hardware and software priorities, with hardware priorities being higher than software priorities. The range of priorities that an SCJVM actually supports may vary between different implementations within these restrictions. To allow the range of supported priorities to be determined in the implementation of the SCJ API, the minimum and maximum hardware and software priority levels can be obtained with the oper-

Operation	Inputs	Outputs	Error Conditions
`getMaxSoftwarePriority`	(none)	priority level	(none)
`getMinSoftwarePriority`	(none)	priority level	(none)
`getNormSoftwarePriority`	(none)	priority level	(none)
`getMaxHardwarePriority`	(none)	priority level	(none)
`getMinHardwarePriority`	(none)	priority level	(none)
`getMainThread`	(none)	thread identifier	(none)
`makeThread`	priority level class identifier method identifier argument list	thread identifier	(none)
`startThreads`	list of thread, backing store and stack identifiers	(none)	invalid identifier thread already started
`getCurrentThread`	(none)	thread identifier	(none)
`suspendThread`	(none)	(none)	thread cannot be blocked thread holds locks
`resumeThread`	thread identifier	(none)	invalid identifier thread not blocked
`setPriorityCeiling`	pointer to object priority level	(none)	invalid priority
`takeLock`	pointer to object	(none)	lock in use
`releaseLock`	pointer to object	(none)	lock not held
`attachInterruptHandler`	interrupt identifier backing store identifier stack identifier class identifier pointer to object	(none)	(none)
`detachInterruptHandler`	interrupt identifier	(none)	(none)
`getInterruptPriority`	interrupt identifier	priority level	(none)
`disableInterrupts`	(none)	(none)	(none)
`enableInterrupts`	(none)	(none)	(none)
`endThread`	(none)	(none)	thread not destroyable thread holds locks

Table 3.2: The operations of the SCJVM scheduler

ations `getMaxSoftwarePriority`, `getMinSoftwarePriority`, `getMaxHardwarePriority`, and `getMinHardwarePriority`. The SCJVM chooses a default normal software priority for threads, that can be queried through the `getNormSoftwarePriority` operation.

Initially there is one thread running, which is called the main thread. The main thread is created when the SCJVM starts and has an implementation-defined priority. The main thread can be suspended by the infrastructure when it is not needed, and resumed when it is needed again (using operations described in the sequel). This allows it to be used for setting up the SCJ application and missions, then suspended during mission execution. The main thread's identifier can be retrieved using the `getMainThread` operation.

Threads other than the main thread can be created by the `makeThread` operation, which takes the entry point and priority level of the thread to be created. The entry point is expressed as the class and identifier of the method that the thread is to run, along with any arguments for the method. This operation returns the identifier of the newly created thread, which must be stored by the infrastructure. The SCJVM does not distinguish between the different thread-release conditions, so for periodic and one-shot threads the infrastructure must set a timer separately using the real-time clock API when a thread is created. The only priorities allowed for threads are the software priorities, as hardware priorities are reserved for interrupts.

The SCJVM threads that are eligible to run must be scheduled as if they are placed in queues with one queue for each priority. At each moment in time, the thread at the front of the highest priority non-empty queue is running. A thread becomes eligible to run after it is started, and stops being eligible to run when it is blocked. Threads are started using the `startThreads` operation, which takes a list of threads to start, together with the backing stores and stacks associated with them. They must be started by the infrastructure when its enclosing mission starts. The reason for the separation between thread creation and thread start is to facilitate the implementation of the SCJ control flow, which requires that threads all start together after mission initialisation has been completely finished. A backing store is provided when a thread is started to serve as the allocation context of the thread, since the per-release memory of an event handler is only created as the handler thread is started. The backing store supplied is only used to set the allocation context in the core execution environment when the thread starts and is not stored by the scheduler.

The identifier of the currently running thread can be obtained through `getCurrentThread`. This operation may be used by the infrastructure as part of obtaining the current schedulable object.

A thread can suspend itself, causing it to become blocked, and be resumed on command from another thread, causing it to become eligible to run again, by the operations `suspendThread` and `resumeThread`. A thread must not be holding any locks when it suspends. These operations are only visible to the program through `wait()` and `notify()` at level 2. These operations are also used in hardware communication, when a thread must wait for the hardware to complete a request, and to implement thread release, whereby a thread remains suspended until released.

The SCJVM must support priority ceiling emulation, which is a mechanism to avoid priority inversion when threads synchronise via locking of objects. In priority ceiling emulation, each object has a priority ceiling, which is the priority of the highest priority thread that may lock the object. When locking an object, a thread's active priority is temporarily raised to the priority ceiling of the object to ensure it is not blocked by higher priority threads waiting to access the same object. This is handled by the `setPriorityCeiling` operation that associates a priority ceiling value to an object. An object that does not have its priority ceiling explicitly set has a priority ceiling equal to the default ceiling. This should be the highest software priority, but it is possible for an SCJVM to have an option to change the default priority ceiling. From our perspective it does not matter what the default priority ceiling is, only that it is a constant value for all threads for a given run of an SCJVM. The SCJVM scheduler does not require a notion of object in order to associate priority ceilings to objects since an object's pointer can be used as an opaque identifier.

The operations for taking and releasing locks are `takeLock` and `releaseLock`. A thread can only take a lock if its active priority and the ceiling priorities of any other objects it holds the locks for are lower than or equal to the ceiling priority of the object the lock is being taken on. Only one thread can take a given object's lock at a time. When a lock is taken, the thread's active priority is raised to the object's priority ceiling. When a thread releases a lock, the thread's active priority is lowered to its previous active priority. The thread may hold nested locks on multiple objects.

The SCJVM scheduler must also manage interrupts, as interrupt handlers must be scheduled along with threads. An interrupt handler can be attached to a given interrupt using the `attach-InterruptHandler` operation, and an interrupt's handler can be removed with the `detach-InterruptHandler` operation. An interrupt with no handler attached to it is ignored. The clock interrupt coming from the hardware is handled by the SCJVM clock (see Section 3.3) and converted into a clock interrupt that is passed to the scheduler for handling by the attached interrupt handler (which should simply call the `triggerAlarm()` method of `Clock`).

Each interrupt has a priority associated with it, which is set by the SCJVM on startup and cannot be changed by the application. These interrupt priorities must be hardware priorities. An interrupt handler is run with the priority of the interrupt it is associated to when that interrupt fires. An interrupt handler interrupts any lower-priority interrupt handlers and any running threads, and blocks lower-priority interrupts from occurring until it has finished. The priority associated with each interrupt can be obtained by the `getInterruptPriority` operation.

Interrupts can be disabled and re-enabled using `disableInterrupts` and `enableInterrupts`. No interrupt handlers can run while interrupts are disabled, but it is implementation-defined as to whether or not interrupts fired while interrupts are disabled are lost.

Finally, the `endThread` operation is used to signal when a thread has reached the end of its execution. This is used for both event handler and interrupt threads. This operation does not automatically destroy the stack or allocation context associated with a thread, which should be removed separately by the infrastructure in the case of event handler threads, and retained for future releases in the case of interrupt handlers. The main thread must not be ended by this operation, since it always exists and is only blocked during mission execution. The end of the main thread corresponds to exit from the SCJVM, which is not considered by this operation. A thread must also not end while it is holding locks, since all locks must be released before the end of the thread is reached. Allowing a thread to end while it is holding locks would prevent resources from ever being freed for use by other threads.

Though the scheduler manages most interrupts, the clock interrupt is managed by the real-time clock, which is the subject of the next section.

3.3 Real-time Clock API

The SCJVM must manage the system real-time clock, providing an interface that allows for the time to be read and alarms to be set to trigger time-based events. The operations of the SCJVM real-time clock are summarised in Table 3.3.

Operation	Inputs	Outputs	Error Conditions
`getSystemTime`	(none)	time	(none)
`getSystemTimePrecision`	(none)	time precision	(none)
`setAlarm`	time	(none)	time in past
`clearAlarm`	(none)	(none)	(none)

Table 3.3: The operations of the SCJVM real-time clock

The main function of the real-time clock API is to provide access to the system time through the `getSystemTime` operation. The SCJ API deals with time values in terms of milliseconds-nanoseconds pairs. That should also be the format for time values passed to and from the SCJVM, though another format could be used. The system time may be measured from January 1, 1970 or from the system start time (in case there is no reliable means of determining the date and time), and so may not correspond to wall-clock time.

The time between ticks of the system clock (its precision) must be made available through the `getSystemTimePrecision` operation. The clock's precision must not change.

The SCJVM must also provide a facility to set an alarm that sends a clock interrupt to the scheduler when a specific time is reached. This facility is provided by the `setAlarm` operation, which accepts an absolute time value at which the alarm should trigger. The time passed to `setAlarm` is required to not be in the past. Running code at a specified relative time offset needs to be handled by the infrastructure. Once an alarm has triggered, it is removed and a new alarm must be set in order to perform events periodically.

The current alarm (if any) can be cleared using the `clearAlarm` operation. Attempting to clear the alarm when there is no alarm set does nothing.

This concludes our discussion of the API of SCJVM services. A formal account of each of the operations in Tables 3.1, 3.2, and 3.3 is the subject of the next section.

3.4 Formal Model

We now present the formal model of the SCJVM services in the *Circus* specification language. The model is structured using a single process for each group of SCJVM services described above, which are then combined in parallel to form a complete model of the SCJVM services. We describe the model of the memory manager in Section 3.4.1, the scheduler in Section 3.4.2, and the real-time clock in Section 3.4.3. Finally, the parts of the model are combined in Section 3.4.4.

3.4.1 Memory Manager

As already said, the SCJVM memory manager is the component that manages the backing stores that underlie memory areas, and provides operations for creating, clearing, and resizing backing stores, and allocating within them. The memory manager also handles allocation and freeing of stack space.

In our formal model, we first declare the types and channels needed for the memory manager model, then build up the model in several layers, beginning with memory blocks that allow operations such as allocation, clearing, and querying of their size, then adding in the structure of backing stores that may contain other backing stores nested inside. Afterwards, the global memory manager covering all the backing stores is specified. Finally, the stack memory management is defined, with the stack area based on the memory blocks model. In this section, we present a *Circus* process that defines the memory manager; the paragraphs of this process include a Z specification that defines each of these layers separately.

Each backing store is identified by an implementation-defined backing store identifier, which may simply be a pointer to the backing store's location in memory. In our model, we define a given set *BackingStoreID* that contains all possible backing store identifiers.

$$[\mathit{BackingStoreID}]$$

The memory allocated by the SCJVM is in the form of raw contiguous blocks of memory. Memory addresses are modelled as natural numbers on the assumption that there are countably many memory addresses.

$$\mathit{MemoryAddress} == \mathbb{N}$$

We use natural number ranges to define the concept of a contiguous memory block, which is central to the formalisation of the requirement that backing stores must not be fragmented.

$$ContiguousMemory == \{ m : \mathbb{P}\, MemoryAddress \mid \exists a, b : MemoryAddress \bullet m = a \mathrel{..} b \}$$

In addition to managing backing stores, the memory manager must also manage stacks, which are also referred to by unique identifiers. The given set, *StackID*, of valid stack identifiers is introduced below.

$$[StackID]$$

We declare channels for each of the operations of the memory manager. Each channel name begins with *MM*, to indicate that it corresponds to an operation of the memory manager API, followed by the name of the service. Operations that return a value have a separate channel to pass that value, the name of which is the name of the service channel with *Ret* appended to it. For example, the channels for `getRootBackingStore` are *MMgetRootBackingStore*, which carries no values as the operation takes no inputs, and *MMgetRootBackingStoreRet*, which communicates backing store identifiers output by the operation. We omit the channel declarations here for the sake of brevity. The definition of all channels and all other definitions we omit here can be found in Appendix A of the extended version of this thesis [13].

Each memory manager function reports a value signalling whether an error occurred and, if so, what error. These error values are of the type *MMReport*, whose definition is sketched below, and are reported over the channel *MMreport*.

$$MMReport ::= MMokay \mid MMoutOfMemory \mid MMnotEmpty \mid \ldots$$

$$\mathbf{channel}\; MMreport : MMReport$$

Lastly, we declare a channel through which the memory manager's initialisation information can be supplied. The initialisation information used by the memory manager is a pair of contiguous memory blocks, representing the space available for backing stores and stacks respectively.

$$\mathbf{channel}\; MMinit : ContiguousMemory \times ContiguousMemory$$

Having declared the channels, we begin the process declaration.

$$\mathbf{process}\; MemoryManager \mathrel{\widehat=} \mathbf{begin}$$

A certain amount of memory overhead can be included in allocated blocks of memory and backing stores to allow for implementation of a memory management algorithm. Memory allocation operations must ensure that there is enough memory available for both the requested amount of memory and the additional overhead. The overhead values must be constant, but may be zero for some memory management algorithms.

$$\mid\; allocationOverhead, backingStoreOverhead : \mathbb{N}$$

We cover each part of the memory manager model in a separate subsection: memory blocks in Section 3.4.1.1, backing stores in Section 3.4.1.2, the global memory manager in Section 3.4.1.3, and stacks in Section 3.4.1.4. Finally, we describe how the Z schemas are lifted to *Circus* operations in Section 3.4.1.5.

3.4.1.1 Memory Blocks

Memory is allocated within memory blocks that keep a record of the amount of *used*, *free*, and *total* memory. Memory blocks form the basis for both backing stores and stack allocation space. It is required that the *used*, *free* and *total* memory are not fragmented. The union of the *used* and *free* memory must also not be fragmented in order for allocation to work correctly. The *used* and *free* memory may not cover all of the memory in the memory block as there may be some overhead, as mentioned above. The *used* and *free* memory must be disjoint.

$$\begin{array}{|l}
\textit{MemoryBlock} \\
\hline
\textit{used}, \textit{free}, \textit{total} : \textit{ContiguousMemory} \\
\hline
\textit{used} \cup \textit{free} \in \textit{ContiguousMemory} \\
\textit{used} \cup \textit{free} \subseteq \textit{total} \\
\textit{used} \cap \textit{free} = \varnothing \\
\hline
\end{array}$$

A memory block must be initialised with the *total* memory covered by the block, including space for any overhead, and initially has no *used* memory. The exact size of the overhead is nondeterministic as this is refined by backing stores and the stack area, which have different overheads.

$$\begin{array}{|l}
\textit{MemoryBlockInit} \\
\hline
\textit{MemoryBlock}' \\
\textit{addresses}? : \textit{ContiguousMemory} \\
\hline
\textit{total}' = \textit{addresses}? \\
\textit{free}' \subseteq \textit{addresses}? \\
\textit{used}' = \varnothing \\
\hline
\end{array}$$

Allocation of memory within memory blocks is performed as described in the *MBAllocate* schema, which takes the requested allocation size as an input, and outputs the allocated contiguous block of memory addresses, *allocated*!. There must be sufficient *free* memory for the requested allocation size. This operation removes *allocated*!, requiring that it be of the given size, from the *free* memory and adds it to the *used* memory, returning the allocated block.

$$\begin{array}{|l}
\textit{MBAllocate} \\
\hline
\Delta \textit{MemoryBlock} \\
\textit{size}? : \mathbb{N} \\
\textit{allocated}! : \textit{ContiguousMemory} \\
\hline
\textit{size}? \leq \# \textit{free} \\
\# \textit{allocated}! = \textit{size}? \\
\textit{allocated}! \subseteq \textit{free} \\
\textit{used}' = \textit{used} \cup \textit{allocated}! \\
\textit{free}' = \textit{free} \setminus \textit{allocated}! \\
\textit{total}' = \textit{total} \\
\hline
\end{array}$$

Since *free* is required to be a *ContinguousMemory*, the removal of *allocated*! is guaranteed not to introduce fragmentation.

The operation of clearing a memory block makes all its *used* memory *free*. This is done by setting the *free* memory to the union of the *used* and *free* memory, and setting the *used* memory to be empty.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBClear}} \\
\hline
\Delta \mathit{MemoryBlock} \\
\hline
\mathit{free}' = \mathit{used} \cup \mathit{free} \\
\mathit{used}' = \varnothing \\
\mathit{total}' = \mathit{total} \\
\hline
\end{array}
$$

There are also operations to read the total, used and free sizes of a memory block. The schemas *MBGetTotalSize*, *MBGetUsedSize* and *MBGetFreeSize*, that define these operations are very simple, since the required information is directly available in the state. So, we have omitted them here.

The operations on the memory manager must be made into robust operations by adding error reporting. Errors are reported by returning a value to indicate the type of error, taken from the *MMReport* type.

The successful completion of an operation is indicated by returning *MMokay*. This is described in the schema *Success*, which is combined with the schemas just defined that describe the successful case of the operations and so does not need to impose any requirements on the state.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Success}} \\
\hline
\mathit{report}! : \mathit{MMReport} \\
\hline
\mathit{report}! = \mathit{MMokay} \\
\hline
\end{array}
$$

The specifications of the error cases follow a common pattern. They all do not change the state and output a value *report*! of type *MMReport* that specifies which error has occurred. Each error case has as its precondition a predicate specifying when the error is triggered. The inputs to the error case are the minimum needed to specify the precondition required. As an example of an error case, we present the schema *MBOutOfMemory* that takes a natural number *size*? as input and reports an out of memory error if *size*? is greater than the amount of free memory.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBOutOfMemory}} \\
\hline
\Xi \mathit{MemoryBlock} \\
\mathit{size}? : \mathbb{N} \\
\mathit{report}! : \mathit{MMReport} \\
\hline
\neg\ \mathit{size}? \leq \#\mathit{free} \\
\mathit{report}! = \mathit{MMoutOfMemory} \\
\hline
\end{array}
$$

Other error cases are defined similarly, so we omit their specifications.

The operations on memory blocks can then be lifted to robust versions. The robust operations are named by prefixing *R* to the name of the lifted operation. Each is formed by taking the conjunction of the operation schema with *Success*, effectively adding an *MMokay* output to the operation; the error cases are placed in disjunction with it. In that way the error cases define what happens when the precondition of the error case is true and the precondition of the operation is false, making the robust operations total since all cases are covered. As an example of a robust operation, we present the robust memory allocation operation, which has *MBOutOfMemory*, defined above, as its only error case.

$$\mathit{RMBAllocate} \;==\; (\mathit{MBAllocate} \land \mathit{Success}) \lor \mathit{MBOutOfMemory}$$

Having modelled memory blocks and the operations upon them, we now proceed to specialise memory blocks to form a model of backing stores. Memory blocks are also used later as the basis for the stack space specification.

3.4.1.2 Backing Stores

The memory manager deals with memory in the form of backing stores, represented by the schema *BackingStore* shown below. *BackingStore* contains two *MemoryBlock*s: an *objectSpace* in which objects are allocated, and a *bsSpace* in which nested backing stores can be allocated. The backing stores nested directly within a backing store, which we refer to as its *children*, are represented by a finite set of backing store identifiers. Backing stores nested deeper are not included in the set of *children*. The full structure of backing store nesting is specified later in the global memory manager. Each backing store in this model also stores its own identifier, *self*. A backing store is required to not be a child of itself. The union of *used* and *free* space in *objectSpace* and *bsSpace* is required to be contiguous, so any overhead must be at the beginning or end of the backing store's space. The *objectSpace* and *bsSpace* must not overlap. The overhead is also specified to be equal in size to *backingStoreOverhead*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BackingStore}} \\ \hline
\mathit{objectSpace} : \mathit{MemoryBlock} \\
\mathit{bsSpace} : \mathit{MemoryBlock} \\
\mathit{children} : \mathbb{F}\, \mathit{BackingStoreID} \\
\mathit{self} : \mathit{BackingStoreID} \\ \hline
\mathit{self} \notin \mathit{children} \\
\mathit{objectSpace}.\mathit{used} \cup \mathit{objectSpace}.\mathit{free} \cup \mathit{bsSpace}.\mathit{used} \cup \mathit{bsSpace}.\mathit{free} \\
\qquad \in \mathit{ContiguousMemory} \\
\mathit{objectSpace}.\mathit{total} \cap \mathit{bsSpace}.\mathit{total} = \varnothing \\
\#(\mathit{objectSpace}.\mathit{used} \cup \mathit{objectSpace}.\mathit{free} \cup \mathit{bsSpace}.\mathit{used} \cup \mathit{bsSpace}.\mathit{free}) \\
\qquad + \mathit{backingStoreOverhead} = \#(\mathit{objectSpace}.\mathit{total} \cup \mathit{bsSpace}.\mathit{total}) \\ \hline
\end{array}$$

Most operations on *BackingStore*s are specified using the Z idiom of promotion in which operations on a local state are lifted to operations over a global state that stores multiple different local states. Promotion works by using the local operation to describe the update of a local state and capturing the local state components using the Z schema binding operator θ. The captured local state is then used to update the global state.

In the case of *BackingStore*, we promote operations on the *objectSpace* and *bsSpace* of a *BackingStore*. We define a promotion schema *PromoteMBsToBS* to perform this. It updates the *objectSpace* of a *BackingStore* according to the state changes of a *MemoryBlock*, distinguished by annotating each state component with $_1$. It also updates the *bsSpace* based on the state changes of a *MemoryBlock* annotated with $_2$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{PromoteMBsToBS} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{BackingStore} \\
\Delta \mathit{MemoryBlock}_1 \\
\Delta \mathit{MemoryBlock}_2 \\
\hline
\theta\, \mathit{MemoryBlock}_1 = \mathit{objectSpace} \\
\theta\, \mathit{MemoryBlock}_2 = \mathit{bsSpace} \\
\mathit{objectSpace}' = \theta\, \mathit{MemoryBlock}'_1 \\
\mathit{bsSpace}' = \theta\, \mathit{MemoryBlock}'_2 \\
\hline
\end{array}
$$

Backing stores are initialised with a contiguous address range *addresses*?, plus the identifier *self*? of the backing store and a natural number *objectSpaceSize*?, which indicates the required size of the *objectSpace*. The *objectSpace* and *bsSpace* of the backing store are initialised as described by *MemoryBlockInit*, with address ranges that partition *addresses*?. The amount of *free* space in the *objectSpace* must be *objectSpaceSize*? and the non-*free* size in *bsSpace* and *objectSpace* must be equal to the *backingStoreOverhead*. The *self* identifier is initialised to the *self*? input and there are initially no *children*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{BackingStoreInit} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{BackingStore}' \\
\mathit{addresses}? : \mathit{ContiguousMemory} \\
\mathit{self}? : \mathit{BackingStoreID} \\
\mathit{objectSpaceSize}? : \mathbb{N} \\
\hline
\exists\, \mathit{objectAddresses}, \mathit{bsAddresses} : \mathit{ContiguousMemory} \bullet \\
\qquad (\exists\, \mathit{MemoryBlock}' \mid \mathit{MemoryBlockInit}[\mathit{objectAddresses}/\mathit{addresses}?] \bullet \\
\qquad\qquad \mathit{objectSpace}' = \theta\, \mathit{MemoryBlock}') \land \\
\qquad (\exists\, \mathit{MemoryBlock}' \mid \mathit{MemoryBlockInit}[\mathit{bsAddresses}/\mathit{addresses}?] \bullet \\
\qquad\qquad \mathit{bsSpace}' = \theta\, \mathit{MemoryBlock}') \land \\
\qquad \mathit{objectAddresses} \cup \mathit{bsAddresses} = \mathit{addresses}? \\
\#\, \mathit{objectSpace}'.\mathit{free} = \mathit{objectSpaceSize}? \\
\#\, \mathit{addresses}? = \#\, \mathit{objectSpace}'.\mathit{free} + \#\, \mathit{bsSpace}'.\mathit{free} + \mathit{backingStoreOverhead} \\
\mathit{self}' = \mathit{self}? \\
\mathit{children}' = \varnothing \\
\hline
\end{array}
$$

The operation of allocating a new child backing store is based on the *MBAllocate* schema. Additional updates to the *BackingStore* state must also be made to set the value of *children*. The identifier of the new child backing store must also be returned and it must be ensured that the total size of the child backing store is large enough to include the *backingStoreOverhead*. These additional requirements are specified in a separate schema *BSAllocateChild*0, which operates over the parent *BackingStore*. It takes an input *size*?, which represents the size of the space to be allocated, and gives an output *childID*!, which is the identifier of the allocated child *BackingStore*. The *size*? must be sufficient to contain the *backingStoreOverhead*. The *childID*! must not be one of the *children* of the parent backing store, nor may it be the identifier *self* of the parent backing store. The *childID*! is added to the *children* set and *self* is unchanged. The other components of the parent *BackingStore* are not constrained by *BSAllocateChild* as they are updated by promoted *MemoryBlock* operations.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{BSAllocateChild0}\;}}\\
\Delta \mathit{BackingStore}\\
\mathit{size}? : \mathbb{N}\\
\mathit{childID}! : \mathit{BackingStoreID}\\
\hline
\mathit{size}? \geq \mathit{backingStoreOverhead}\\
\mathit{childID}! \notin \mathit{children} \land \mathit{childID}! \neq \mathit{self}\\
\mathit{children}' = \mathit{children} \cup \{\mathit{childID}!\}\\
\mathit{self}' = \mathit{self}\\
\hline
\end{array}$$

The *BSAllocateChild* schema is combined with an *MBAllocate* operation promoted to act over the *bsSpace* of the *BackingStore* using the *PromoteMBsToBS* schema. The input and output of *MBAllocate* are renamed to remove the decoration applied to make *MBAllocate* act over the *bsSpace*. The *objectSpace* is unaffected by this operation.

$$\begin{array}{l}
\mathit{BSAllocateChild} == \exists\,(\Xi \mathit{MemoryBlock})_1;\ (\Delta \mathit{MemoryBlock})_2 \bullet \mathit{BSAllocateChild0}\\
\qquad \land\ \mathit{MBAllocate}_2[\mathit{size}?/\mathit{size}?_2, \mathit{allocated}!/\mathit{allocated}!_2] \land \mathit{PromoteMBsToBS}
\end{array}$$

The other *BackingStore* operations are specified in a similar fashion, promoting *MemoryBlock* operations to act upon the *objectSpace* and *bsSpace* of a *BackingStore*, and specifying additional conditions in a separate schema.

Clearing a backing store removes all of its *children* and does not affect its *self* identifier, as specified in *BSClear*0 below.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{BSClear0}\;}}\\
\Delta \mathit{BackingStore}\\
\hline
\mathit{children}' = \varnothing\\
\mathit{self}' = \mathit{self}\\
\hline
\end{array}$$

The operation of clearing a backing store is then specified by *BSClear*, which is a combination of *BSClear*0 with two *MBClear* operations promoted to act over both *objectSpace* and *bsSpace*.

$$\begin{array}{l}
\mathit{BSClear} == \exists\,\Delta \mathit{MemoryBlock}_1;\ \Delta \mathit{MemoryBlock}_2 \bullet\\
\qquad \mathit{BSClear0} \land \mathit{MBClear}_1 \land \mathit{MBClear}_2 \land \mathit{PromoteMBsToBS}
\end{array}$$

Allocating object memory within a backing store is performed with the additional inputs and output defined in *BSAllocate*0. There is an input *size*?, which is the required size of the object memory to be allocated, and the allocated memory is provided via an output *allocated*!. Since space for the *allocationOverhead* must be allocated when object memory is allocated, an *actualSize* value is computed by adding the *allocationOverhead* to *size*?. The *children* and *self* components of the *BackingStore* are unaffected by this operation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;\mathit{BSAllocate0}\;}}\\
\Delta \mathit{BackingStore}\\
\mathit{size}? : \mathbb{N}\\
\mathit{allocated}! : \mathit{ContiguousMemory}\\
\mathit{actualSize} : \mathbb{N}\\
\hline
\mathit{actualSize} = \mathit{size}? + \mathit{allocationOverhead}\\
\mathit{children}' = \mathit{children}\\
\mathit{self}' = \mathit{self}\\
\hline
\end{array}$$

The allocation operation is then specified by *BSAllocate* below, which promotes *MBAllocate* to act over the *objectSpace* of the *BackingStore*. The *actualSize* is used as the *size*? input to *MBAllocate* and hidden so that the *size*? input to *BSAllocate*0 is the only input of *BSAllocate*.

$$
\begin{aligned}
&\mathit{BSAllocate} == \exists\, \Delta \mathit{MemoryBlock}_1;\ \Xi \mathit{MemoryBlock}_2;\ \mathit{actualSize} : \mathbb{N} \bullet \\
&\quad \mathit{BSAllocate}0 \land \mathit{MBAllocate}_1[\mathit{actualSize}/\mathit{size}?_1, \mathit{allocated}!/\mathit{allocated}!_1] \\
&\qquad \land \mathit{PromoteMBsToBS}
\end{aligned}
$$

The operation of resizing a backing store adjusts the sizes of its *objectSpace* and *bsSpace*, and so it is specified in its own schema, *BSResize*, rather than being promoted from a *MemoryBlock* operation. There is one input to *BSResize*, which is *newSize*?, the desired new size of the *objectSpace*. This operation may be used when there is no *used* memory in *objectSpace* (as is the case when resizing mission memory after mission termination, or resizing private memory when reentering it), or when *bsSpace* is empty (as is the case for immortal memory when the SCJVM starts up, and for mission memory when the mission object is created). In both cases, there must be no *used* memory in *bsSpace*. The *newSize*? must be sufficient to include any existing *used* memory in *objectSpace* (which is always true in the first case, where it is empty). The *objectSpace* is resized so that the combination of its *used* and *free* space is as large as *newSize*. The *used* space in *objectSpace* must remain the same, so only the *free* space can change. The additional *free* space in *objectSpace* is taken from the *free* space of *bsSpace*. The *used* memory in *bsSpace* remains empty after the operation. As a consequence of the *used* part of *bsSpace* being empty, *children* must also be empty, since there can be no child backing stores. The union of the *total* space in *bsSpace* and *objectSpace* remains the same, although the *backingStoreOverhead* may move between the *objectSpace* and *bsSpace* (it may not change position, but it may be counted as part of a different set in order to preserve the invariant of *BackingStore*). The *self* identifier is unaffected.

BSResize

$$
\begin{aligned}
&\Delta \mathit{BackingStore} \\
&\mathit{newSize}? : \mathbb{N} \\
\hline
&\mathit{objectSpace}.\mathit{used} = \varnothing \lor \mathit{bsSpace}.\mathit{free} = \varnothing \\
&\mathit{bsSpace}.\mathit{used} = \varnothing \\
&\mathit{newSize}? \geq \#\, \mathit{objectSpace}.\mathit{used} \\
&\#(\mathit{objectSpace}'.\mathit{used} \cup \mathit{objectSpace}'.\mathit{free}) = \mathit{newSize}? \\
&\mathit{objectSpace}'.\mathit{used} = \mathit{objectSpace}.\mathit{used} \\
&\mathit{objectSpace}'.\mathit{free} \cup \mathit{bsSpace}'.\mathit{free} = \mathit{objectSpace}.\mathit{free} \cup \mathit{bsSpace}.\mathit{free} \\
&\mathit{bsSpace}'.\mathit{used} = \varnothing \\
&\mathit{children} = \varnothing = \mathit{children}' \\
&\mathit{bsSpace}'.\mathit{total} \cup \mathit{objectSpace}'.\mathit{total} = \mathit{bsSpace}.\mathit{total} \cup \mathit{objectSpace}.\mathit{total} \\
&\mathit{self}' = \mathit{self}
\end{aligned}
$$

The other operations on backing stores are defined by promoting the memory block operations to operate on the *objectSpace* or *bsSpace* of a backing store, keeping the set of *children* and the *self* identifier the same.

$$
\begin{array}{l}
\mathit{BSGetTotalSize} == [\Delta \mathit{BackingStore} \mid \mathit{children}' = \mathit{children} \land \mathit{self}' = \mathit{self}] \land \\
\quad \exists\, \Delta \mathit{MemoryBlock}_1;\ \Xi \mathit{MemoryBlock}_2 \bullet \\
\qquad \mathit{MBGetTotalSize}_1[\mathit{size}!/\mathit{size}!_1] \land \mathit{PromoteMBsToBS} \\
\mathit{BSGetUsedSize} == [\Delta \mathit{BackingStore} \mid \mathit{children}' = \mathit{children} \land \mathit{self}' = \mathit{self}] \land \\
\quad \exists\, \Delta \mathit{MemoryBlock}_1;\ \Xi \mathit{MemoryBlock}_2 \bullet \\
\qquad \mathit{MBGetUsedSize}_1[\mathit{size}!/\mathit{size}!_1] \land \mathit{PromoteMBsToBS} \\
\mathit{BSGetFreeSize} == [\Delta \mathit{BackingStore} \mid \mathit{children}' = \mathit{children} \land \mathit{self}' = \mathit{self}] \land \\
\quad \exists\, \Delta \mathit{MemoryBlock}_1;\ \Xi \mathit{MemoryBlock}_2 \bullet \\
\qquad \mathit{MBGetFreeSize}_1[\mathit{size}!/\mathit{size}!_1] \land \mathit{PromoteMBsToBS} \\
\mathit{BSGetRemainingBS} == [\Delta \mathit{BackingStore} \mid \mathit{children}' = \mathit{children} \land \mathit{self}' = \mathit{self}] \land \\
\quad \exists\, \Xi \mathit{MemoryBlock}_1;\ \Delta \mathit{MemoryBlock}_2 \bullet \\
\qquad \mathit{MBGetFreeSize}_2[\mathit{size}!/\mathit{size}!_2] \land \mathit{PromoteMBsToBS}
\end{array}
$$

These operations must then be made into robust operations that report error values if their preconditions are not met. Some of the error reporting schemas for memory blocks can be reused, but there are new preconditions in the backing store operations based on the memory block operations that must be accounted for. These require additional schemas, but we have omitted their definitions here as they are similar in form to the memory block error cases.

Using these schemas, the operations on backing stores can be made into robust operations. This lifting to robust operations is similar to that for the memory block operations. As an example, we present the robust backing store initialisation operation. The initialisation schema presented earlier is combined with *Success* to output *MMokay* in the event of a successful initialisation. Its only error case is that in which the provided set of addresses is too small to contain the backing store overhead. As the schema for this error case is used for other operations, it has an initial state that is not present during initialisation and so it must be hidden using existential quantification.

$$
\begin{array}{l}
\mathit{RBackingStoreInit} == \\
\quad (\mathit{BackingStoreInit} \land \mathit{Success}) \lor (\exists\, \mathit{BackingStore} \bullet \mathit{BSSizeTooSmall})
\end{array}
$$

We omit the other robust operations as they are similar to the robust memory block operations.

This concludes our model of backing stores as individual structures. Next we specify the global memory manager, which contains all backing stores and whose invariant records the relations between them.

3.4.1.3 Global Memory Manager

The memory manager must hold information on all backing stores and the identifier of the one that is the root backing store. All backing stores must be nested within the root backing store.

The information about all the backing stores is held in the the global memory manager state, which we split into several parts to make specification of invariants easier. The first part is *GlobalStoresManager*, which contains a map, *stores*, from backing store identifiers to backing stores. This map is partial, since not all backing store identifiers may be used, and finite, since there will only ever be a finite number of backing stores in use. This is because none of the operations on the memory manager creates an infinite number of backing stores. The

GlobalStoresManager also contains the identifier of the root backing store, *rootBackingStore*, since that is used in specifying several of the invariants of the memory manager.

$$
\begin{array}{|l}
\textit{GlobalStoresManager} \\
\hline
\textit{stores} : \textit{BackingStoreID} \text{ ⤀ } \textit{BackingStore} \\
\textit{rootBackingStore} : \textit{BackingStoreID} \\
\hline
\textit{rootBackingStore} \in \operatorname{dom} \textit{stores} \\
\forall\, \textit{bsid} : \operatorname{dom} \textit{stores} \bullet \\
\qquad (\textit{stores bsid}).\textit{self} = \textit{bsid} \land \\
\qquad (\textit{stores bsid}).\textit{children} \subseteq \operatorname{dom} \textit{stores} \land \\
\qquad (\lambda\, \textit{childID} : (\textit{stores bsid}).\textit{children} \bullet \\
\qquad\qquad (\textit{stores childID}).\textit{objectSpace}.\textit{total} \cup (\textit{stores childID}).\textit{bsSpace}.\textit{total}) \\
\qquad\qquad\qquad \text{partition } (\textit{stores bsid}).\textit{bsSpace}.\textit{used} \\
\hline
\end{array}
$$

The first invariant of the *GlobalStoresManager* state requires that the *rootBackingStore* identifier be in the domain of *stores*. The remaining invariants of the *GlobalStoresManager* are specified to hold for any backing store identifier *bsid* in the domain of *stores*. The *self* identifier of the backing store *bsid* is mapped to under *stores* must be the same as *bsid* itself. This ensures that a backing store identifier cannot be mapped to a completely different backing store and imposes an injectivity condition on *stores* whereby two backing store identifiers cannot be mapped to the same backing store. The *children* of the backing store identified by *bsid* must also be in the domain of *stores*. Finally, for each *childID* in the *children* set for the backing store denoted by *bsid*, the backing store memory corresponding to *childID* must be in the *used bsSpace* memory and distinct from that of other child backing stores.

Then, the *GlobalMemoryManager* schema represents the full state of the backing stores in the memory manager. It contains *GlobalStoresManager* and also contains a relation, *childRelation*, that represents the structure of backing store nesting, relating backing store identifiers to the identifiers of their children.

$$
\begin{array}{|l}
\textit{GlobalMemoryManager} \\
\hline
\textit{GlobalStoresManager} \\
\textit{childRelation} : \textit{BackingStoreID} \leftrightarrow \textit{BackingStoreID} \\
\hline
\forall\, \textit{bsid} : \operatorname{dom} \textit{stores} \bullet \textit{childRelation} \,(\!|\, \{\textit{bsid}\} \,|\!) = (\textit{stores bsid}).\textit{children} \\
\operatorname{dom} \textit{stores} = (\textit{childRelation}^{*}) \,(\!|\, \{\textit{rootBackingStore}\} \,|\!) \\
\forall\, \textit{bsid} : \operatorname{dom} \textit{stores} \bullet \textit{bsid} \notin \textit{childRelation}^{+} \,(\!|\{\textit{bsid}\}\,|\!) \\
\hline
\end{array}
$$

The first invariant of *GlobalMemoryManager* defines *childRelation* by stating that the image of a given backing store identifier, *bsid*, under *childRelation* is the *children* set of the corresponding backing store. The remaining two invariants restrict the structure of *childRelation* to that of a tree. The first requires every identifier in the domain of *stores* to be reachable from the *rootBackingStore* by stating that the image of *rootBackingStore* under the reflexive transitive closure of *childRelation* must be equal to the domain of *stores*. The second ensures there can be no loops by stating that each backing store cannot be related to itself under the transitive closure of *childRelation*.

We note that the invariants of *GlobalMemoryManager* and *GlobalStoresManager* are sufficient to ensure distinctness of backing stores. The invariant of *GlobalStoresManager* ensures that a backing store's identifier in *stores* must be the same as its *self* identifier, and the invariant of

BackingStore then ensures it cannot be the same as any of its *children*. Also, the invariant of *GlobalStoresManager* requires that the total memory space occupied by each of a backing store's *children* must partition the *used* memory in its *bsSpace*. This ensures that the child memory areas cannot overlap, and, since the *used* memory in *bsSpace* is part of the total memory of the backing store, this applies transitively to the children's children. Since the invariant of *GlobalMemoryManager* requires all backing stores to be a child of the root backing store, each backing store may only overlap with its (direct or indirect) parents or children. This thus prevents two (non-nested) backing stores from sharing the same child (directly or indirectly), since then those backing stores would both contain the space occupied by the child and thus would overlap. They are not permitted to overlap, since they must both be children of a common parent since they are at least (possibly indirect) children of the root backing store.

Initially there must be one backing store provided, which is the root backing store. The memory manager must be initialised with the set of memory addresses to be used for the root backing store. The size of the object space for the root backing store is initially the entire size of the provided addresses, minus space for the *backingStoreOverhead*. The root backing store is initialised with these addresses as described in *RBackingStoreInit*, with the input *self*? set to the *rootBackingStore* identifier, which may be any available backing store identifier. The *childRelation* is initially empty because there is only one backing store that initially has no children.

GlobalMemoryManagerInit

$GlobalMemoryManager'$
$addresses? : ContiguousMemory$
$report! : MMReport$

$\exists\, objectSpaceSize? : \{\#\, addresses? - backingStoreOverhead\} \bullet$
$\exists\, BackingStore' \mid RBackingStoreInit[rootBackingStore'/self?] \bullet$
$\quad stores' = \{rootBackingStore' \mapsto \theta\, BackingStore'\}$
$childRelation' = \varnothing$

The operations on the global memory manager are defined by promoting operations on backing stores, updating the backing stores in the *stores* map. This is handled by the *PromoteBS* schema, defined below, which takes a backing store identifier as input. The state components for both the local state and the global state are brought into scope so that the promotion can act on both of them. The update of the local state is performed by the operation schema, which is combined with the promotion schema. The backing store identifier is required to be in the domain of *stores* and the initial state of the corresponding backing store is captured from the global state. The final state of the local operation is then captured and used to update *stores*.

PromoteBS

$\Delta GlobalMemoryManager$
$\Delta BackingStore$
$bs? : BackingStoreID$

$bs? \in \operatorname{dom} stores$
$\theta\, BackingStore = stores\ bs?$
$stores' = stores \oplus \{bs? \mapsto \theta\, BackingStore'\}$
$rootBackingStore' = rootBackingStore$

As an example of a promotion, we present the operation of allocating memory. The local state used in the promotion is hidden using existential quantification so that the operation is an operation on the global state. The operation over the local state is combined with the promotion schema declared above to form the operation over the global state. We also adjust the operation to return the address of the start of the memory block. This is achieved by conjoining another schema to it that contains an *address*! output, which is set to be the minimum of the *allocated*! addresses output by *RBSAllocate*.

$$
\begin{array}{l}
\mathit{GlobalAllocateMemory} == \\
\quad \exists \Delta \mathit{BackingStore};\ \mathit{allocated}! : \mathit{ContiguousMemory} \bullet \mathit{RBSAllocate} \land \mathit{PromoteBS} \land \\
\qquad [\mathit{allocated}! : \mathit{ContiguousMemory};\ \mathit{address}! : \mathit{MemoryAddress} \mid \\
\qquad\quad \mathit{address}! = \mathit{min}\ \mathit{allocated}!]
\end{array}
$$

The other promoted operations are the operations for getting the total, used and free object space size of a backing store: *GlobalGetTotalSize*, *GlobalGetUsedSize* and *GlobalGetFreeSize*, and the operation for getting the free space in the backing store space of a backing store, *GlobalGetRemainingBS*. These are defined similarly, so we omit their definitions here.

Some of the global memory manager operations differ from the standard form for promotion as they need to promote more than one schema at once or update the global state in an unusual way. Those operations are explained here.

The operation of making a new backing store inside a given backing store is performed by allocating space inside the parent backing store, as specified by *RBSAllocateChild*, and initialising the new child backing store, as specified by *RBackingStoreInit*. The schema that describes this operation, *GlobalMakeBS*, is defined below by promoting both of these operations.

$$
\begin{array}{|l}
\textit{GlobalMakeBS} \\
\hline
\Delta \mathit{GlobalMemoryManager} \\
\mathit{size}? : \mathbb{N} \\
\mathit{objectSpaceSize}? : \mathbb{N} \\
\mathit{parentID}? : \mathit{BackingStoreID} \\
\mathit{childID}! : \mathit{BackingStoreID} \\
\hline
\mathit{parentID}? \in \mathrm{dom}\, \mathit{stores} \\
\exists \mathit{actualSize} : \mathbb{N} \mid \mathit{actualSize} = \mathit{size}? + \mathit{backingStoreOverhead} \bullet \\
\exists \mathit{allocated}! : \mathit{ContiguousMemory} \bullet \\
\exists \mathit{Parent}, \mathit{child} : \mathit{BackingStore} \bullet \\
\quad (\exists \Delta \mathit{BackingStore};\ \mathit{report}! : \mathit{MMReport} \mid \\
\qquad \mathit{RBSAllocateChild}[\mathit{actualSize}/\mathit{size}?] \bullet \\
\qquad \theta\, \mathit{BackingStore} = \mathit{stores}\ \mathit{parentID}? \land \\
\qquad \mathit{Parent} = \theta\, \mathit{BackingStore}' \land \\
\qquad \mathit{childID}! \notin \mathrm{dom}\, \mathit{stores} \land \\
\qquad \mathit{report}! = \mathit{MMokay}) \land \\
\quad (\exists \mathit{BackingStore}';\ \mathit{report}! : \mathit{MMReport} \mid \\
\qquad \mathit{RBackingStoreInit}[\mathit{allocated}!/\mathit{addresses}?, \mathit{childID}!/\mathit{self}?] \bullet \\
\qquad \mathit{self}' = \mathit{childID}! \land \\
\qquad \mathit{child} = \theta\, \mathit{BackingStore}' \land \\
\qquad \mathit{report}! = \mathit{MMokay}) \land \\
\quad \mathit{stores}' = \mathit{stores} \oplus \{\mathit{parentID}? \mapsto \mathit{Parent}, \mathit{childID}! \mapsto \mathit{child}\} \\
\mathit{rootBackingStore}' = \mathit{rootBackingStore} \\
\hline
\end{array}
$$

The *GlobalMakeBS* schema takes as input the required size of the new backing store, *size*? and the identifier of its parent, *parentID*?. The identifier of the new child backing store, *childID*!, is given as output. There is a precondition that the parent identifier must be in the domain of *stores*, since it must be a valid backing store identifier. We also define a value *actualSize* to be *size*? plus *backingStoreOverhead*, so that the *size*? is the actual amount of usable space in the backing store, without any overhead. A local variable *allocated*! is brought into scope using existential quantification to hold the addresses output by *RBSAllocateChild*. Two local variables *Parent* and *child* are also introduced to store the final local states of the promoted operations. The promotions of each of the local state operations are then specified. The operation *RBSAllocateChild* is promoted to act on the local state of the parent, with *actualSize* replacing its size input, and its final state is stored in *Parent*. The *childID*! and *allocated*! variables are identified with the outputs from *RBSAllocateChild* of the same names. It is required that *childID*! not be already in the domain of *stores*. The error report from the promoted operations must be a report of success for the global operation to work; the cases where it is not are handled as separate error cases. The operation *RBackingStoreInit* is promoted to initialise a local state for the newly created backing store. The outputs *allocated*! and *childID*! from *RBSAllocateChild* are used to replace the *addresses*? and *self*? inputs to *RBackingStoreInit*. The new backing store is stored in *child*. The *stores* map is updated to contain *Parent* and *child*.

The operation of clearing a backing store is described by the schema *GlobalClearBS*, which takes a backing store identifier, *toClear*?, as input and promotes the *RBSClear* operation to act over the corresponding backing store. The *toClear*? identifier is required to be a valid backing store identifier. The error report is required to be a report of success, as for the promotions in the *GlobalMakeBS* schema above. This promotion differs from that described by *PromoteBS* in that it removes backing stores nested within the cleared backing store from the *stores* map. The identifiers of the nested backing stores are those reachable via the transitive closure of the child relation, so we define a set *reachable* as the image of the cleared backing store's identifier under the transitive closure of the child relation. This set of backing store identifiers is removed from the domain of *stores* using the domain antirestriction operator, $\lhd$, before the local state of the cleared backing store is used to update *stores*. The child relation is also updated, with the nested backing stores removed from its range using the range antirestriction operator, $\rhd$.

```
── GlobalClearBS ──────────────────────────────
│ ΔGlobalMemoryManager
│ toClear? : BackingStoreID
├──────────
│ toClear? ∈ dom stores
│ ∃ ΔBackingStore; report! : MMReport | RBSClear •
│      θ BackingStore = stores toClear? ∧
│      report! = MMokay ∧
│      ∃ reachable : 𝔽 BackingStoreID |
│            reachable = childRelation⁺ ⦇{toClear?}⦈ •
│            stores′ = (reachable ⩤ stores) ⊕ {toClear? ↦ θ BackingStore′} ∧
│            childRelation′ = childRelation ⩥ reachable
│ rootBackingStore′ = rootBackingStore
└───────────────────────────────────────────────
```

It must also be possible to determine which backing store a given memory address belongs to, in order to implement the `getMemoryArea` method of `MemoryArea`. This is handled by the *GlobalFindAddress* schema, which does not affect the local state of any backing stores or the state of the memory manager. The address to search for is taken as an input, *address*?, to the operation and must be within the *objectSpace* for some backing store for this to work. That backing store is unique, since *address*? must be in the *bsSpace* for parent backing stores, so we obtain it using the Z unique specification operator, μ, returning it as the *backingStore*! output.

$$\begin{array}{l}
\textit{GlobalFindAddress} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{address}? : \textit{MemoryAddress} \\
\textit{backingStore}! : \textit{BackingStoreID} \\
\hline
\exists\, \textit{bsid} : \operatorname{dom} \textit{stores} \bullet \textit{address}? \in (\textit{stores}\ \textit{bsid}).\textit{objectSpace}.\textit{total} \\
\textit{backingStore}! = (\mu\, \textit{bsid} : \operatorname{dom} \textit{stores} \mid \textit{address}? \in (\textit{stores}\ \textit{bsid}).\textit{objectSpace}.\textit{total}) \\
\hline
\end{array}$$

There must also be an operation to obtain the identifier of the root backing store from the global memory manager. This is provided by the schema *GlobalGetRootBackingStore*, which we omit here as it just provides the value of the state component *rootBackingStore*.

These operations on the global memory manager are made into robust operations that report errors. The robust versions of the backing store operations are used in specifying some of the robust global memory manager operations. Some new error reporting schemas are also required to handle errors that can occur at the level of the global memory manager and errors in the reports from promoted schemas. As the structure of making the operations robust is similar to that used for memory blocks and backing stores, we omit it here.

This concludes the Z definition of backing store operations; they are lifted to *Circus* actions after the definition of the stack management operations, presented next.

3.4.1.4 Stack Memory Manager

As previously discussed, in addition to providing facilities for backing stores and memory allocation, the SCJVM must allow for allocating thread stacks. The stacks should be allocated in an area separate from the root backing store, set aside for the allocation of stacks when the SCJVM starts. The SCJVM memory management need only provide the memory for the stacks; management of the stack contents must be handled by the core execution environment.

The stack area is a memory block that holds additional information about allocated stacks so that they can be deallocated when the thread is removed. So, thread stacks may have their own memory overhead associated with them.

$$\mid\ \textit{stackOverhead} : \mathbb{N}$$

The stack memory manager controls a memory block using a function, *stacks*, mapping stack identifiers to the memory of the associated stack. The memory allocated for stacks must partition the used stack memory.

$$\begin{array}{l}
\textit{StackMemoryManager} \\
\hline
\textit{MemoryBlock} \\
\textit{stacks} : \textit{StackID} \pfun \textit{ContiguousMemory} \\
\hline
\textit{stacks} \text{ partition } \textit{used} \\
\hline
\end{array}$$

The stack manager is initialised with a given area of memory for allocating stacks. The initialisation schema is based on the initialisation of memory blocks, with *addresses*? renamed to *stackSpace*?. There are initially no stacks allocated, so *stacks* is empty.

$$
\begin{array}{|l}
\textit{StackMemoryManagerInit} \\
\hline
\textit{StackMemoryManager}' \\
\textit{MemoryBlockInit}[\textit{stackSpace}?/\textit{addresses}?] \\
\hline
\textit{stacks}' = \varnothing \\
\hline
\end{array}
$$

The operation to create a new stack of a given size is defined by *StackCreate* and is based on *RMBAllocate*. The stack overhead must be taken into account in this operation as we are allocating stacks, not space for objects or backing stores. The new stack's identifier, *newStack*! must be one not already in use. The new identifier is stored in the map *stacks*, mapping it to the allocated memory, and is also output from the operation.

$$
\begin{array}{|l}
\textit{StackCreate} \\
\hline
\Delta \textit{StackMemoryManager} \\
\textit{size}? : \mathbb{N} \\
\textit{newStack}! : \textit{StackID} \\
\hline
\textit{newStack}! \notin \operatorname{dom} \textit{stacks} \\
\exists\, \textit{actualSize} : \mathbb{N} \mid \textit{actualSize} = \textit{size}? + \textit{stackOverhead} \bullet \\
\exists\, \textit{report}! : \textit{MMReport};\ \textit{allocated}! : \textit{ContiguousMemory} \bullet \\
\qquad \textit{RMBAllocate}[\textit{actualSize}/\textit{size}?] \land \textit{report}! = \textit{MMokay} \land \\
\qquad \textit{stacks}' = \textit{stacks} \oplus \{\textit{newStack}! \mapsto \textit{allocated}!\} \\
\hline
\end{array}
$$

There is also an operation to delete a stack, freeing the memory used for it. This is defined by the schema *StackDelete*, which takes the identifier of the stack to delete as input. The identifier is required to be an existing valid identifier, i.e. in the domain of *stacks*. The space allocated for the stack is removed from the used memory and added to the free memory. The free memory with the stack allocation added to it must be contiguous. The identifier of the deleted stack is removed from the domain of *stacks*.

$$
\begin{array}{|l}
\textit{StackDelete} \\
\hline
\Delta \textit{StackMemoryManager} \\
\textit{stack}? : \textit{StackID} \\
\hline
\textit{stack}? \in \operatorname{dom} \textit{stacks} \\
\textit{used}' = \textit{used} \setminus \textit{stacks}\ \textit{stack}? \\
\textit{free}' = \textit{free} \cup \textit{stacks}\ \textit{stack}? \in \textit{ContiguousMemory} \\
\textit{stacks}' = \{\textit{stack}?\} \ntriangleleft \textit{stacks} \\
\textit{total}' = \textit{total} \\
\hline
\end{array}
$$

The stack memory manager operations are made into robust operations.

So far, we have defined the memory manager operations as a Z data model. The operations must now be made available via the *Circus* channels for the memory manager process.

3.4.1.5 Memory Manager Operations

We make the operations defined by the schemas above available as services accessible via *Circus* channels. Each of the services described in Section 3.1 are provided. The correspondence between the services described in section Section 3.1, the *Circus* actions described here, and the Z schemas defined earlier is shown in Table 3.4.

Service	*Circus* action	Z schema
`getRootBackingStore`	*GetRootBackingStore*	*RGlobalGetRootBackingStore*
`getTotalSize`	*GetTotalSize*	*RGlobalGetTotalSize*
`getUsedSize`	*GetUsedSize*	*RGlobalGetUsedSize*
`getFreeSize`	*GetFreeSize*	*RGlobalGetFreeSize*
`getRemainingBackingStore`	*GetRemainingBS*	*RGlobalGetRemainingBS*
`findBackingStore`	*FindBackingStore*	*RGlobalFindAddress*
`allocateMemory`	*AllocateMemory*	*RGlobalAllocateMemory*
`makeBackingStore`	*MakeBackingStore*	*RGlobalMakeBS*
`clearBackingStore`	*ClearBackingStore*	*RGlobalClearBS*
`resizeBackingStore`	*ResizeBackingStore*	*RGlobalResizeBS*
`createStack`	*CreateStack*	*RStackCreate*
`destroyStack`	*DestroyStack*	*RStackDestroy*

Table 3.4: The relationship between the memory manager services and the *Circus* actions and Z schemas defining them

The state of the memory manager process is made up of both the global memory manager and the stack memory manager.

$$\mathbf{state}\ \mathit{GlobalMemoryManager} \land \mathit{StackMemoryManager}$$

The memory manager is initialised by taking the root backing store and stack space as inputs and using the initialisation schemas for both the global memory manager and the stack memory manager. The error value from the global memory manager initialisation is reported.

$$\begin{array}{l}
\mathit{Init} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{MMReport} \bullet \\
\quad \mathit{MMinit}?\mathit{addresses}?\mathit{stackSpace} \longrightarrow \\
\qquad \big(\mathit{RGlobalMemoryManagerInit} \land \mathit{StackMemoryManagerInit}\big); \\
\quad \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip}
\end{array}$$

The lifting of operations to *Circus* actions follows a common pattern, which can be seen here in the definition of the *GetRootBackingStore* action. The request to perform the operation, along with any inputs, is received on the operation's channel. The operation is then performed as specified by a corresponding schema and any outputs from the operation are communicated on the return channel for the operation. The error report is communicated on the error reporting channel before the operation ends.

$$\begin{array}{l}
\mathit{GetRootBackingStore} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{MMReport};\ \mathit{rbs} : \mathit{BackingStoreID} \bullet \\
\quad \mathit{MMgetRootBackingStore} \longrightarrow \big(\mathit{RGlobalGetRootBackingStore}\big); \\
\quad \mathit{MMgetRootBackingStoreRet}!\mathit{rbs} \longrightarrow \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip}
\end{array}$$

The memory manager continuously presents all its operations in a loop. Any operation can be chosen once the previous operation has completed.

$$Loop \mathrel{\widehat=} GetRootBackingStore \mathrel{\Box} GetCurrentAllocationContext \mathrel{\Box} \cdots \; ; \; Loop$$

The main action of the memory manager process first requires initialisation and then enters the operation loop declared above.

$\bullet\ Init \; ; \; Loop$

end

This concludes the specification of the memory manager. We have built the memory manager in several layers, first defining the concept of a memory block, in which allocations can occur and which is used as the basis for specifying backing stores and the stack space. We then specified backing stores, which are pairs of memory blocks that keep a record of other backing stores nested within them. The backing store operations have then been promoted to act over a global memory manager with a view of all backing stores. Allocation and deallocation of space for stacks has also been specified, with the stack space treated as a memory block to allow memory for stacks to be allocated within it. Finally, we have lifted the operations to *Circus* actions, making them available over channels, via which the inputs to the operation (if any) are provided. Outputs from operations with output are provided via a separate return channel and all operations also report whether an error occurred via a separate error reporting channel.

Having specified the SCJVM services related to memory management in this section, we cover the next group of services, relating to scheduling, in the next section.

3.4.2 Scheduler

The SCJVM scheduler must manage separate threads of execution, which involves tracking information about threads, selecting which thread to run, handling locks, and blocking threads. The scheduler must also manage interrupts as they interfere with thread scheduling.

Threads are identified by unique implementation-defined thread identifiers of the *ThreadID* type.

$$[ThreadID]$$

There are two particular *ThreadID* values that identify special threads that exist from the start of the program. These are *idle*, which identifies the idle thread that does nothing and runs when no other thread is available to run, and *main*, which identifies the thread used during SCJVM startup. These two identifiers must be distinct.

$$\begin{array}{|l}
idle, main : ThreadID \\
\hline
idle \neq main
\end{array}$$

Threads are scheduled according to their priorities. Priorities are divided into hardware priorities, which are used for interrupt handlers, and software priorities, which are used for threads.

There must be support for at least 28 priorities, with hardware priorities being higher than software priorities. One software priority must be designated as the normal priority.

$$
\begin{array}{|l}
minHwPriority, maxHwPriority : \mathbb{N} \\
minSwPriority, maxSwPriority : \mathbb{N} \\
normSwPriority : \mathbb{N} \\
\hline
(maxHwPriority - minHwPriority + 1) + (maxSwPriority - minSwPriority + 1) \geq 28 \\
minSwPriority < maxSwPriority < minHwPriority < maxHwPriority \\
minSwPriority \leq normSwPriority \leq maxSwPriority
\end{array}
$$

We define separate types for thread priorities and interrupt priorities so it can be checked in the model that a thread is not started with an interrupt priority.

$$ThreadPriority == minSwPriority \mathinner{..} maxSwPriority$$
$$InterruptPriority == minHwPriority \mathinner{..} maxHwPriority$$

For the situations where either a thread or interrupt priority could be used, we use a type formed by joining the two sets.

$$Priority == ThreadPriority \cup InterruptPriority$$

The threads represent threads of execution of Java bytecode programs in the core execution environment. The scheduler must be able to inform the core execution environment when a thread switch occurs so that it can swap the stack and program counter. This is done using the *CEEswitchThread* channel.

$$\mathbf{channel}\ CEEswitchThread : ThreadID \times ThreadID$$

If there are no thread switches that need to be handled, and the program reaches a point at which a thread switch may occur, the scheduler signals to the program to proceed with execution on the *CEEproceed* channel.

$$\mathbf{channel}\ CEEproceed : ThreadID$$

The scheduler must also be able to provide the information required when a thread starts, which is the thread's initial backing store, class, method and arguments. To declare the appropriate channel, the types of class identifiers, method identifiers and virtual machine words are required. We declare the types of class and method identifiers as the given types *ClassID* and *MethodID*

$$[ClassID, MethodID]$$

The type, *Word*, of virtual machine words is defined to be the type of integers, since words are signed for the purposes of arithmetic.

$$Word == \mathbb{Z}$$

The information is communicated to the core execution environment via the *CEEstartThread* channel.

$$\begin{array}{l}\mathbf{channel}\ CEEstartThread \\ \quad : ThreadID \times BackingStoreID \times StackID \times ClassID \times MethodID \times \mathrm{seq}\ Word\end{array}$$

While the concept of objects is mainly handled by the core execution environment, the scheduler must have some notion of object identifiers in order to manage locks on objects, so we define them here. This is provided by the *ObjectID* type, which may simply represent an opaque pointer to the object. Object identifiers are drawn from the same space as memory addresses, since they represent the location of objects in memory.

$$ObjectID == MemoryAddress$$

We distinguish one *ObjectID* value as the *null* object identifier, representing the absence of an object.

$$\mid \; null : ObjectID$$

We also define a function *WordToObjectID* to convert between *Word* values and *ObjectID* values, since a machine word can be interpreted as a pointer to an object, as well as a signed integer value for arithmetic.

$$\mid \; WordToObjectID : Word \rightarrowtail\!\!\!\!\!\!\!+\;\, ObjectID$$

We leave the structure of objects represented by *ObjectID*s to the core execution environment.

The SCJVM scheduler also manages interrupts, which also have unique identifiers. The precise set of identifiers will likely depend on what interrupt vectors the hardware offers.

$$[InterruptID]$$

Hardware interrupts are received via the *HWinterrupt* channel, which communicates the interrupt identifier,

$$\mathbf{channel}\; HWinterrupt : InterruptID$$

The hardware is also required to permit enabling and disabling interrupts. This is represented in the model by the channels *HWenableInterrupts* and *HWdisableInterrupts*.

$$\mathbf{channel}\; HWenableInterrupts, HWdisableInterrupts$$

Although it is mainly left implementation-defined which interrupts are offered, it is required that there is a clock interrupt that is fired at regular intervals. This interrupt is not directly handled by the scheduler but is instead used by the real-time clock described in the next section.

$$\mid \; clockInterrupt : InterruptID$$

When the real-time clock has an alarm trigger, it passes on the clock interrupt to the scheduler to run a handler for it. In this model that is represented by the *RTCclockInterrupt* channel.

$$\mathbf{channel}\; RTCclockInterrupt$$

The SCJVM scheduler offers services made available through channels. Similarly to the memory manager channels, the channels are named after the service names given in Table 3.2 prefixed with *S* to indicate that they are handled by the scheduler. Services with both inputs and outputs have an additional return channel, named with the suffix *Ret*. Services that provide an output and take no inputs simply have one channel on which output is communicated. For brevity we do not include the full channel list here.

As with the memory manager operations, each operation of the scheduler reports whether or not an error occurred and, if so, what error. These error values are of type *SReport* and are reported over the channel *Sreport*.

$$SReport ::= Sokay \mid SnonexistentThread \mid SthreadAlreadyStarted \mid \cdots$$

$$\mathbf{channel}\ Sreport : SReport$$

With the channels and datatypes declared, we begin the process declaration.

$$\mathbf{process}\ Scheduler \mathrel{\widehat=} \mathbf{begin}$$

We cover each part of the scheduler model in a separate subsection: information about threads in Section 3.4.2.1, the priority scheduler in Section 3.4.2.2, priority ceiling emulation in Section 3.4.2.3, and interrupt handling in Section 3.4.2.4. Then, we handle some considerations around communicating thread switches to the core execution environment in Section 3.4.2.5. Finally, we describe how the Z schemas are lifted to *Circus* operations in Section 3.4.2.6.

3.4.2.1 Threads

The SCJVM scheduler manages threads and stores information about them. The thread information stored by the scheduler is represented in the *ThreadInfo* schema, defined below. The scheduler stores the class, identifier and arguments for the initial method executed by each thread. Each thread also has a base and current priority, which may change due to the priority ceiling emulation system described later. Each of these pieces of thread information is represented via a partial function from thread identifiers to the type of the information and all the functions are required to have the same domain. It is required that the current priority is not less than the base priority as the priority can only be temporarily raised, not lowered.

ThreadInfo

$$\begin{array}{l}
threadClass : ThreadID \pfun ClassID \\
threadMethod : ThreadID \pfun MethodID \\
threadArgs : ThreadID \pfun \operatorname{seq} Word \\
basePriority : ThreadID \pfun Priority \\
currentPriority : ThreadID \pfun Priority \\
\hline
\operatorname{dom} threadClass = \operatorname{dom} threadMethod = \operatorname{dom} threadArgs = \\
\qquad \operatorname{dom} currentPriority = \operatorname{dom} basePriority \\
\forall t : \operatorname{dom} currentPriority \bullet currentPriority\ t \geq basePriority\ t
\end{array}$$

Because a lot of operations, particularly those involved in priority ceiling emulation, only change the *currentPriority*, we define an additional schema, *PreserveThreadInfo*, that specifies that all components of *ThreadInfo* except *currentPriority* remain the same.

$$PreserveThreadInfo == \Xi ThreadInfo \setminus (currentPriority, currentPriority')$$

We also define an operation *RemoveThreadInfo*, that removes an input identifier, *thread*? from the domain of all the functions in *ThreadInfo*.

RemoveThreadInfo

$$\begin{array}{l}
\Delta \mathit{ThreadInfo} \\
\mathit{thread?} : \mathit{ThreadID} \\
\hline
\mathit{threadClass}' = \{\mathit{thread?}\} \ndres \mathit{threadClass} \\
\mathit{threadMethod}' = \{\mathit{thread?}\} \ndres \mathit{threadMethod} \\
\mathit{threadArgs}' = \{\mathit{thread?}\} \ndres \mathit{threadArgs} \\
\mathit{basePriority}' = \{\mathit{thread?}\} \ndres \mathit{basePriority} \\
\mathit{currentPriority}' = \{\mathit{thread?}\} \ndres \mathit{currentPriority}
\end{array}$$

SCJVM threads may be in one of several states at any given time. The information on which threads are in each state is represented as specified in the *ThreadManager* schema, which contains sets of identifiers recording the threads that are in each of these states. An SCJVM thread may be either *created* and waiting to start, *started* but not running (because a higher priority thread is running), *blocked* or the currently running thread, *current*. There is only a single *current* thread, so it is represented by a single identifier, whereas multiple threads may be in the other states, so they are represented as sets of threads. In addition to these states, there is also a set of *free* thread identifiers. The *idle* thread must not be in any of the sets *free*, *created*, *started* or *blocked* (though it may be the *current* thread).

The thread states partition the space of thread identifiers into *free* identifiers, *created* but not started threads, *started* but not running threads, *blocked* threads, and the *current* or *idle* threads. The *main* thread must be either *started*, *blocked* or the *current* thread, since it cannot be destroyed and there is never a time when it is not started.

ThreadManager

$$\begin{array}{l}
\mathit{free}, \mathit{created}, \mathit{started}, \mathit{blocked} : \mathbb{P}\ \mathit{ThreadID} \\
\mathit{current} : \mathit{ThreadID} \\
\hline
\langle \mathit{free}, \mathit{created}, \mathit{started}, \mathit{blocked}, \{\mathit{current}, \mathit{idle}\} \rangle \text{ partition } \mathit{ThreadID} \\
\mathit{main} \in \mathit{started} \cup \mathit{blocked} \cup \{\mathit{current}\}
\end{array}$$

Initially all thread identifiers are in the *free* set, except the ones used for the *idle* and *main* threads. The *current* thread is initially *main* and there are no threads in the other states.

ThreadManagerInit

$$\begin{array}{l}
\mathit{ThreadManager}\,' \\
\hline
\mathit{free}' = \mathit{ThreadID} \setminus \{\mathit{idle}, \mathit{main}\} \\
\mathit{current}' = \mathit{main} \\
\mathit{created}' = \mathit{started}' = \mathit{blocked}' = \varnothing
\end{array}$$

Because most operations only affect some sets in *ThreadManager*, we define schemas to specify that only certain sets change. They are named using *Change* followed by the names of the components permitted to change. Since they are similar, we only present the first one here.

$$\mathit{ChangeFreeCreated} == \Xi\,\mathit{ThreadManager} \setminus (\mathit{free}, \mathit{free}', \mathit{created}, \mathit{created}')$$

Having defined all the relevant information concerning threads, we now describe how they are scheduled according to their priorities.

3.4.2.2 Priority Scheduler

The SCJVM scheduler is a preemptive priority scheduler, which stores queues of thread identifiers for each priority. We define these queues and the operations upon them separately. A queue is represented using a Z sequence. We take the front of the sequence to be the back of the queue to ensure the correct ordering of queue elements when the priority queues are flattened into a single queue. The sequence is taken to be injective since no thread identifier can occur more than once in the same queue.

$$Queue == \operatorname{iseq} ThreadID$$

We define operations *pushFront* and *pushBack* to push identifiers onto a given queue. As mentioned above, the first element of the sequence is taken to be the last element of the queue, so *pushFront* pushes to the back of the sequence and *pushBack* pushes to the front of the sequence. An extra function, *pushBackSet*, is used to push all of a finite set of thread identifiers to the front of a queue in a nondeterministic order.

$$\begin{array}{|l}
pushFront, pushBack : ThreadID \to Queue \to Queue \\
pushBackSet : \mathbb{F}\ ThreadID \to Queue \to Queue \\
\hline
\forall thread : ThreadID;\ queue : Queue \bullet \\
\quad pushFront\ thread\ queue = queue \frown \langle thread \rangle \land \\
\quad pushBack\ thread\ queue = \langle thread \rangle \frown queue \\
\forall threads : \mathbb{F}\ ThreadID;\ queue : Queue \bullet \\
\quad \exists threadSequence : \operatorname{iseq} ThreadID \mid \operatorname{ran} threadSequence = threads \bullet \\
\qquad pushBackSet\ threads\ queue = threadSequence \frown queue
\end{array}$$

We also provide operations *queueFront* and *removeFromQueue* to obtain the identifier at the front of the queue and remove an identifier from a queue. The *queueFront* operation is simply the *last* operation for taking last element of the sequence, and the *removeFromQueue* operation uses the Z filtering operator, $\upharpoonright$, to filter the identifier out of the queue.

$$\begin{array}{|l}
queueFront : Queue \nrightarrow ThreadID \\
removeFromQueue : ThreadID \to Queue \to Queue \\
\hline
queueFront = last \\
\forall thread : ThreadID;\ queue : Queue \bullet \\
\quad removeFromQueue\ thread\ queue = queue \upharpoonright \{thread\}
\end{array}$$

The state of the scheduler is defined in the *Scheduler* schema, which contains all the components of *ThreadInfo* and *ThreadManager*. The scheduler also contains a queue of thread identifiers for each priority. We represent the priority queues by a function from *Priority* to the *Queue* type defined above. There is additionally a set of thread identifiers that identify threads executing interrupt handlers. The identifiers in the priority queues must be all the identifiers of the *started* threads and the identifier of the *current* thread, but not the identifier of the *idle* thread, even if it is the *current* thread. This is because the *idle* thread is only selected to run if there are no other threads available and so does not fit into the normal ordering of threads. The identifiers in two different priority queues are required to be disjoint, since a thread cannot have two different priorities. A related requirement is that each thread identified in the priority queues has its current priority the same as the priority of its queue. The functions defined in *ThreadInfo* are related to the states defined in *ThreadManager* by requiring that the domain of *currentPriority* (and hence all the other functions) be the union of all the thread sets except *free*. The interrupt threads must be within the *started* and *current* threads, since interrupt threads cannot self suspend and are created as needed.

Scheduler

$$
\begin{array}{l}
\mathit{ThreadInfo} \\
\mathit{ThreadManager} \\
\mathit{priorityQueues} : \mathit{Priority} \rightarrow \mathit{Queue} \\
\mathit{interruptThreads} : \mathbb{P}\ \mathit{ThreadID} \\
\hline
\bigcup\{ q : \operatorname{ran} \mathit{priorityQueues} \bullet \operatorname{ran} q \} = (\mathit{started} \cup \{\mathit{current}\}) \setminus \{\mathit{idle}\} \\
\operatorname{disjoint} (\lambda\, p : \mathit{Priority} \bullet \operatorname{ran} (\mathit{priorityQueues}\ p)) \\
\forall\, p : \mathit{Priority} \bullet \forall\, t : \operatorname{ran}(\mathit{priorityQueues}\ p) \bullet \mathit{currentPriority}\ t = p \\
\operatorname{dom} \mathit{currentPriority} = \mathit{created} \cup \mathit{started} \cup \mathit{blocked} \cup \{\mathit{current}, \mathit{idle}\} \\
\mathit{interruptThreads} \subseteq \mathit{started} \cup \{\mathit{current}\}
\end{array}
$$

Since operations usually only need to update one priority queue, we provide a function to simplify such updates.

$$
\begin{array}{l}
\mathit{updatePriorityQueue} \\
\qquad : \mathit{Priority} \rightarrow (\mathit{Queue} \rightarrow \mathit{Queue}) \rightarrow (\mathit{Priority} \rightarrow \mathit{Queue}) \rightarrow (\mathit{Priority} \rightarrow \mathit{Queue}) \\
\hline
\forall\, \mathit{priority} : \mathit{Priority};\ f : \mathit{Queue} \rightarrow \mathit{Queue};\ \mathit{pqs} : \mathit{Priority} \rightarrow \mathit{Queue} \bullet \\
\qquad \mathit{updatePriorityQueue}\ \mathit{priority}\ f\ \mathit{pqs} = \mathit{pqs} \oplus \{\mathit{priority} \mapsto f\ (\mathit{pqs}\ \mathit{priority})\}
\end{array}
$$

The initialisation of the scheduler is as specified in the schema *SchedulerInit*, defined below. The thread states, together with the *main* and *idle* thread identifiers are initialised as described by the schema *ThreadManagerInit*. The *main* thread has a priority supplied as an input to the initialisation and the *idle* thread has the lowest possible priority. These are initially used for both the base and current priorities. The initial values of the other components of *ThreadInfo* do not matter since they are only used for starting a thread. The queue for the *main* thread's priority initially contains only the *main* thread's identifier and all other priority queues are empty. There are initially no interrupt threads.

SchedulerInit

$$
\begin{array}{l}
\mathit{Scheduler}' \\
\mathit{ThreadManagerInit} \\
\mathit{mainPriority}? : \mathit{ThreadPriority} \\
\hline
\mathit{currentPriority}' = \mathit{basePriority}' = \\
\qquad \{\mathit{main} \mapsto \mathit{mainPriority}?, \mathit{idle} \mapsto \mathit{minSwPriority}\} \\
\mathit{priorityQueues}'\ \mathit{mainPriority}? = \langle \mathit{main} \rangle \\
\forall\, p : \mathit{Priority} \mid p \neq \mathit{mainPriority}? \bullet \mathit{priorityQueues}'\ p = \langle\rangle \\
\mathit{interruptThreads}' = \varnothing
\end{array}
$$

When a new thread is created, its class, method, arguments and priority are stored. The operation of creating a new thread is defined by the schema *ThreadCreate*, which takes this information as input. The new thread is given an identifier from the *free* identifier set; that identifier is an output from the operation. The functions in *ThreadInfo* are updated to include the new thread identifier and the information input to the operation. The priority input to the operation is used for both the current and base priority. The new thread's identifier is removed from the *free* identifiers and added to the *created* set. The other thread states are unaffected and the interrupt threads do not change as this operation is not used to create interrupt threads.

$$
\begin{array}{|l}
\textit{ThreadCreate} \\
\hline
\Delta \textit{Scheduler} \\
\textit{ChangeFreeCreated} \\
\textit{class}? : \textit{ClassID};\ \textit{method}? : \textit{MethodID};\ \textit{args}? : \mathrm{seq}\ \textit{Word} \\
\textit{priority}? : \textit{ThreadPriority} \\
\textit{newID}! : \textit{ThreadID} \\
\hline
\textit{newID}! \in \textit{free} \\
\textit{threadClass}' = \textit{threadClass} \oplus \{\textit{newID}! \mapsto \textit{class}?\} \\
\textit{threadMethod}' = \textit{threadMethod} \oplus \{\textit{newID}! \mapsto \textit{method}?\} \\
\textit{threadArgs}' = \textit{threadArgs} \oplus \{\textit{newID}! \mapsto \textit{args}?\} \\
\textit{currentPriority}' = \textit{currentPriority} \oplus \{\textit{newID}! \mapsto \textit{priority}?\} \\
\textit{basePriority}' = \textit{basePriority} \oplus \{\textit{newID}! \mapsto \textit{priority}?\} \\
\textit{free}' = \textit{free} \setminus \{\textit{newID}!\} \land \textit{created}' = \textit{created} \cup \{\textit{newID}!\} \\
\textit{interruptThreads}' = \textit{interruptThreads} \land \textit{priorityQueues}' = \textit{priorityQueues} \\
\hline
\end{array}
$$

Although creating a new thread does not change the *current* thread, many of the remaining scheduler operations do update the *current* thread. Since the method for picking the *current* thread is the same for each operation, we separate its specification into its own schema, *PickNewCurrent*, which modifies the state of *Scheduler*. It takes an input, *newStarted*?, which represents the contents of the *started* set and at the point where this operation is used. It outputs an identifier, *previous*!, which is the identifier of the previous *current* thread, before it is changed.

The new *current* thread is chosen as the thread at the front of the highest priority thread queue. This is specified by first flattening the priority queues using a function *flattenQueues*, which squashes *priorityQueues* into a sequence and concatenates the queues together into a single queue. Since this is just a combination of standard Z functions grouped together for convenience, we omit its definition here. The *idle* thread is pushed to the back of the flattened queue to ensure it is non-empty, and the front of the queue is then chosen as the new *current* thread. The final *priorityQueues* component is used for this, as the priority queues may be modified before a new *current* thread is chosen.

The new *started* set is obtained by adding the old *current* thread to *newStarted*?, while removing the new *current* thread from it, along with the *idle* thread, to ensure that they remain outside the set. The other state components are left unspecified, since this schema is intended to be used as part of other schemas, which specify their values.

$$
\begin{array}{|l}
\textit{PickNewCurrent} \\
\hline
\Delta \textit{Scheduler} \\
\textit{newStarted}? : \mathbb{F}\ \textit{ThreadID} \\
\textit{previous}! : \textit{ThreadID} \\
\hline
\textit{current}' = \textit{queueFront}\ (\textit{pushBack}\ \textit{idle}\ (\textit{flattenQueues}\ \textit{priorityQueues}')) \\
\textit{started}' = (\textit{newStarted}? \cup \{\textit{current}\}) \setminus \{\textit{current}', \textit{idle}\} \\
\textit{previous}! = \textit{current} \\
\hline
\end{array}
$$

When a set of threads is started, they are added to the back of the queues for their priorities. This is described by the schema *ThreadStarts*, which takes as input a set, *toStart*?, containing the identifiers of the threads to be started. All these threads are required to be in the *created* set to ensure they have been created but not yet started. The identifiers of the threads are removed from the *created* set and, for each priority, the identifiers of the threads in *toStart*? having that priority are added to the corresponding priority queue using the *pushBackSet* function. The newly started threads are added to the *started* set, which is then further updated as specified by *PickNewCurrent*, which chooses a new *current* thread. The *previous*! identifier output from *PickNewCurrent* is output in *ThreadStarts*. The other state components remain unaffected.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ThreadStarts}} \\
\Delta \mathit{Scheduler};\ \Xi \mathit{ThreadInfo};\ \mathit{ChangeCreatedStartedCurrent} \\
\mathit{toStart}? : \mathbb{F}\ \mathit{ThreadID} \\
\mathit{previous}! : \mathit{ThreadID} \\
\hline
\mathit{toStart}? \subseteq \mathit{created} \\
\mathit{created}' = \mathit{created} \setminus \mathit{toStart}? \\
\forall p : \mathrm{dom}\ \mathit{priorityQueues} \bullet \mathit{priorityQueues}'\ p = \\
\qquad \mathit{pushBackSet}\ ((\mathit{currentPriority}^{\sim})\ (\!|\ \{p\}\ |\!))\ (\mathit{priorityQueues}\ p) \\
\exists\ \mathit{newStarted}? : \{\mathit{started} \cup \mathit{toStart}?\} \bullet \mathit{PickNewCurrent} \\
\mathit{interruptThreads}' = \mathit{interruptThreads} \\
\hline
\end{array}$$

Destruction of a thread is handled by the *ThreadDestroy* schema, which takes the identifier of the thread to be destroyed as input. The thread identifier is required not to be in *free*, or to be the *main* or *idle* thread. The thread also cannot be an interrupt thread, since this operation is for destroying non-interrupt threads. Interrupt threads must be handled differently, and are considered later in this section.

The thread's identifier is removed from the domain of all the thread information functions, removing the information about it from the scheduler's state as specified by *RemoveThreadInfo*. The thread is added to the *free* thread set, and removed from the *created*, *started* and *blocked* threads. The *thread*? identifier is removed from its priority queue by the *removeFromQueue* operation. A new *current* thread is chosen as specified by *PickNewCurrent*, with *started* modified to exclude the destroyed thread (since *PickNewCurrent* may add it into *started* if it is *current*), and the *previous*! value is returned from the operation. The interrupt threads are unaffected.

$$
\begin{array}{l}
\textit{ThreadDestroy} \\
\hline
\Delta \textit{Scheduler} \\
\textit{RemoveThreadInfo} \\
\textit{thread}? : \textit{ThreadID} \\
\textit{previous}! : \textit{ThreadID} \\
\hline
\textit{thread}? \notin \textit{free} \cup \{\textit{idle}, \textit{main}\} \cup \textit{interruptThreads} \\
\textit{free}' = \textit{free} \cup \{\textit{thread}?\} \\
\textit{created}' = \textit{created} \setminus \{\textit{thread}?\} \land \textit{blocked}' = \textit{blocked} \setminus \{\textit{thread}?\} \\
\exists \textit{priority} : \{\textit{currentPriority thread}?\} \bullet \textit{priorityQueues}' = \\
\quad\quad \textit{updatePriorityQueue priority } (\textit{removeFromQueue thread}?) \textit{ priorityQueues} \\
\exists \textit{newStarted}? : \{\textit{started} \setminus \{\textit{thread}?\}\};\ \textit{started}0 : \mathbb{F}\ \textit{ThreadID} \bullet \\
\quad\quad \textit{PickNewCurrent}[\textit{started}0/\textit{started}'] \land \textit{started}' = \textit{started}0 \setminus \{\textit{thread}?\} \\
\textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

An SCJVM thread may be suspended, causing it to pause running and block. This operation is defined by the schema *ThreadSuspend*, which does not affect the thread information and takes the identifier, *toSuspend*?, of the thread to be suspended as an input.

The *toSuspend*? thread must be either *current* or one of the *started* threads. It must not be the *idle* thread or an interrupt thread since those threads cannot be suspended. The *toSuspend*? thread is added to the *blocked* thread set and filtered out from the threads in its priority queue as in *DestroyThread*. A new *current* thread is chosen, as described by *PickNewCurrent*, much as in *DestroyThread*. The other state components remain unaffected.

$$
\begin{array}{l}
\textit{ThreadSuspend} \\
\hline
\Delta \textit{Scheduler} \\
\Xi \textit{ThreadInfo} \\
\textit{toSuspend}? : \textit{ThreadID} \\
\textit{previous}! : \textit{ThreadID} \\
\hline
\textit{toSuspend}? \in \{\textit{current}\} \cup \textit{started} \\
\textit{toSuspend}? \notin \{\textit{idle}\} \cup \textit{interruptThreads} \\
\textit{blocked}' = \textit{blocked} \cup \{\textit{toSuspend}?\} \\
\exists \textit{priority} : \{\textit{currentPriority toSuspend}?\} \bullet \textit{priorityQueues}' = \\
\quad \textit{updatePriorityQueue priority } (\textit{removeFromQueue toSuspend}?) \textit{ priorityQueues} \\
\exists \textit{newStarted}? : \{\textit{started} \setminus \{\textit{toSuspend}?\}\};\ \textit{started}0 : \mathbb{F}\ \textit{ThreadID} \bullet \\
\quad \textit{PickNewCurrent}[\textit{started}0/\textit{started}'] \land \textit{started}' = \textit{started}0 \setminus \{\textit{toSuspend}?\} \\
\textit{free}' = \textit{free} \land \textit{created}' = \textit{created} \land \textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

A suspended thread remains suspended until it is signalled to resume by the operation defined by *ThreadResume*. This operation does not affect the thread information and takes as input the identifier of the thread to be resumed, which must be one that is blocked. Its identifier is removed from the *blocked* set and added to the *started* set. The resumed thread is placed at the back of the queue for its current priority. A new *current* thread is then chosen as specified in *PickNewCurrent*. The other state components are unaffected.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ThreadResume}} \\
\hline
\Delta \mathit{Scheduler} \\
\mathit{ChangeBlockedStartedCurrent} \\
\Xi \mathit{ThreadInfo} \\
\mathit{thread?} : \mathit{ThreadID} \\
\mathit{previous!} : \mathit{ThreadID} \\
\hline
\mathit{thread?} \in \mathit{blocked} \\
\mathit{blocked}' = \mathit{blocked} \setminus \{\mathit{thread?}\} \\
\exists\, \mathit{priority} : \{\mathit{currentPriority}\ \mathit{thread?}\} \bullet \mathit{priorityQueues}' = \\
\qquad \mathit{updatePriorityQueue}\ \mathit{priority}\ (\mathit{pushBack}\ \mathit{thread?})\ \mathit{priorityQueues} \\
\exists\, \mathit{newStarted?} : \{\mathit{started} \cup \{\mathit{thread?}\}\} \bullet \mathit{PickNewCurrent} \\
\mathit{interruptThreads}' = \mathit{interruptThreads} \\
\hline
\end{array}
$$

It should be noted that SCJ programs do not have direct access to the functionality of suspending and resuming threads. It is provided for the infrastructure to implement, for example, device access and mission initialisation.

We have specified threads and how they are scheduled, but the interactions between threads when taking the lock on an object must also be specified. This is covered next, where we describe the priority ceiling emulation policy that SCJ requires for locking.

3.4.2.3 Priority Ceiling Emulation

The SCJVM must support priority ceiling emulation and locking of objects. This is accounted for by *PCEScheduler*, which is an extension of the *Scheduler* state to include information required for priority ceiling emulation and locking. The state contains a function *priorityCeiling* that associates a ceiling priority to each object identifier. This function is total as the scheduler does not need to be aware of which objects actually exist and can simply assign a ceiling priority to all the possible identifiers, locking on those passed to it from outside. There are also functions *lockHolder* and *lockCount* that map object identifiers to the identifier of the thread that holds each object's lock and the number of times each lock has been taken (a thread may retake the lock on an object it has already locked, forming multiple nested locks). These functions are partial as it only makes sense to hold this information for an object that has been locked. For convenience, we also have a function, *locksHeld*, mapping threads to the sets of objects they hold locks for, which may be empty.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PCEScheduler}} \\
\hline
\mathit{Scheduler} \\
\mathit{priorityCeiling} : \mathit{ObjectID} \rightarrow \mathit{Priority} \\
\mathit{lockHolder} : \mathit{ObjectID} \pfun \mathit{ThreadID} \\
\mathit{lockCount} : \mathit{ObjectID} \pfun \mathbb{N}_1 \\
\mathit{locksHeld} : \mathit{ThreadID} \rightarrow \mathbb{P}\ \mathit{ObjectID} \\
\hline
\mathrm{dom}\ \mathit{lockCount} = \mathrm{dom}\ \mathit{lockHolder} \\
\mathrm{ran}\ \mathit{lockHolder} \subseteq \mathit{started} \cup \{\mathit{current}\} \\
\forall\, t : \mathit{ThreadID} \bullet \mathit{locksHeld}\ t = \mathit{lockHolder}^{\sim}\ (\!| \{t\} |\!) \\
\forall\, t : \mathit{ThreadID} \bullet \mathit{currentPriority}\ t = \\
\qquad\qquad \mathit{max}\ (\{\mathit{basePriority}\ t\} \cup \{\ o : \mathit{locksHeld}\ t \bullet \mathit{priorityCeiling}\ o\ \}) \\
\hline
\end{array}
$$

The domains of the functions *lockCount* and *lockHolder* are required to be the same, and the range of *lockHolder* is required to be within the started and current thread identifiers since those are the only threads that can hold locks. The *locksHeld* function is defined to map to the relational image of the inverse of *lockHolder* for a given thread. The current priority of a thread under priority ceiling emulation is given by the maximum of the thread's base priority and the priority ceilings of all the objects it holds locks on.

The *PCEScheduler* is initialised as for *Scheduler*, with additional initialisation of the state components introduced in *PCEScheduler*. Initially all objects have the default priority ceiling, which is the maximum software priority. The *lockHolder* and *lockCount* maps are empty, since no locks are initially held. The state invariant defining *locksHeld* is sufficient to uniquely determine it so an explicit initialisation is not provided for it here.

$$
\begin{array}{|l}
\hline
\textit{PCESchedulerInit} \\
\quad \textit{PCEScheduler}' \\
\quad \textit{SchedulerInit} \\
\hline
\quad \forall x : \textit{ObjectID} \bullet \textit{priorityCeiling}'\ x = \textit{maxSwPriority} \\
\quad \textit{lockHolder}' = \varnothing \\
\quad \textit{lockCount}' = \varnothing \\
\hline
\end{array}
$$

Taking the lock on an object is specified by considering two cases: the case in which the lock is not held and the case in which an object is attempting to retake a lock it already holds. The handling of the first case is described by the schema *PCETakeLock*, defined below, which takes as input the identifier of the object to lock and the thread taking the lock. The object is required to not be in the domain of *lockHolder* for this case to ensure it does not already have a thread locking it, and the current priority of the thread must also be less than the object's priority ceiling. The object is added to *lockHolder*, associated with the given thread, and also to *lockCount*, associated with 1, since this is the first time the thread has taken the lock on this object. The current priority of the thread is set to the maximum of the thread's current priority and the priority ceiling of the object. The priority queues are updated by first removing the thread's identifier from the queue for its old priority, then adding it to the front of the queue for its new priority. The *current* thread is updated as specified by *PickCurrentThread*. The other state components are unchanged.

$$
\begin{array}{|l}
\hline
\textit{PCETakeLock} \\
\quad \Delta \textit{PCEScheduler};\ \textit{ChangeStartedCurrent};\ \textit{PreserveThreadInfo} \\
\quad \textit{PickNewCurrent}[\textit{started} / \textit{newStarted}?] \\
\quad \textit{object}? : \textit{ObjectID} \\
\quad \textit{thread}? : \textit{ThreadID} \\
\hline
\quad \textit{object}? \notin \operatorname{dom} \textit{lockHolder} \\
\quad \textit{currentPriority}\ \textit{thread}? \leq \textit{priorityCeiling}\ \textit{object}? \\
\quad \textit{lockHolder}' = \textit{lockHolder} \oplus \{\textit{object}? \mapsto \textit{thread}?\} \\
\quad \textit{lockCount}' = \textit{lockCount} \oplus \{\textit{object}? \mapsto 1\} \\
\quad \textit{currentPriority}' = \textit{currentPriority} \oplus \\
\qquad \{\textit{thread}? \mapsto \textit{max}\ \{\textit{currentPriority}\ \textit{thread}?, \textit{priorityCeiling}\ \textit{object}?\}\} \\
\quad \exists\, \textit{priority} : \{\textit{currentPriority}\ \textit{thread}?\} \bullet \textit{priorityQueues}' = \\
\qquad (\textit{updatePriorityQueue}\ \textit{priority}\ (\textit{removeFromQueue}\ \textit{thread}?) \mathbin{⨾} \\
\qquad \textit{updatePriorityQueue}\ \textit{priority}\ (\textit{pushFront}\ \textit{thread}?))\ \textit{priorityQueues} \\
\quad \textit{priorityCeiling}' = \textit{priorityCeiling} \land \textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

The second case, where a thread already holds the lock, is described by *PCERetakeLock*, which is similar to *PCETakeLock* in that it takes as input the identifier of the object to lock and the thread taking the lock. In this case, the object is required to be in the domain of *lockHolder* but it must map to the input thread identifier, since a thread cannot take a lock held by another thread. The *lockCount* value for the object is incremented by one and the values for other objects are unchanged. All other state components are unchanged. A new *current* thread is not chosen, since the information that affects the choice of *current* thread is not changed, but *current* is output as *previous*! for consistency with the interface of the *PCETakeLock* case.

$$
\begin{array}{|l}
\hline
\quad \mathit{PCERetakeLock} \\
\Delta \mathit{PCEScheduler} \\
\Xi \mathit{Scheduler} \\
\mathit{object}? : \mathit{ObjectID} \\
\mathit{thread}? : \mathit{ThreadID} \\
\mathit{previous}! : \mathit{ThreadID} \\
\hline
\mathit{object}? \in \operatorname{dom} \mathit{lockHolder} \\
\mathit{lockHolder}\ \mathit{object}? = \mathit{thread}? \\
\mathit{lockCount}'\ \mathit{object}? = \mathit{lockCount}\ \mathit{object}? + 1 \\
\{\mathit{object}?\} \lhd \mathit{lockCount}' = \{\mathit{object}?\} \lhd \mathit{lockCount} \\
\mathit{priorityCeiling}' = \mathit{priorityCeiling} \\
\mathit{lockHolder}' = \mathit{lockHolder} \land \mathit{locksHeld}' = \mathit{locksHeld} \\
\mathit{previous}! = \mathit{current} \\
\hline
\end{array}
$$

The operation of releasing the lock on an object is similarly split into two cases: the case in which the lock is held only once, and the case in which a thread holds multiple nested locks on the same object. The first case is described by *PCEReleaseLock*, defined below, which takes as input the identifier of the object to be unlocked and the thread holding the lock. The object is required to be in the set of locks held by the thread and *lockCount* must be 1. The object's identifier is removed from the domain of *lockHolder* and *lockCount* since it is no longer locked by any thread. The current priority of the thread is set to the maximum of the thread's base priority and the priority ceilings of any other objects it holds locks on. The thread is placed at the front of the priority queue for its new priority and a new *current* thread is selected, in the same way as in *PCETakeLock*. The other state components are unaffected.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{PCEReleaseLock} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{PCEScheduler};\ \mathit{ChangeStartedCurrent};\ \mathit{PreserveThreadInfo} \\
\mathit{PickNewCurrent}[\mathit{started} / \mathit{newStarted}?] \\
\mathit{object}? : \mathit{ObjectID} \\
\mathit{thread}? : \mathit{ThreadID} \\
\hline
\mathit{object}? \in \mathit{locksHeld}\ \mathit{thread}? \\
\mathit{lockCount}\ \mathit{object}? = 1 \\
\mathit{lockHolder}' = \{\mathit{object}?\} \lhd \mathit{lockHolder} \\
\mathit{lockCount}' = \{\mathit{object}?\} \lhd \mathit{lockCount} \\
\mathit{currentPriority}' = \mathit{currentPriority} \oplus \{\mathit{thread}? \mapsto \mathit{max} \\
\qquad (\{o : \mathit{locksHeld}'\ \mathit{thread}? \bullet \mathit{priorityCeiling}\ o\} \cup \{\mathit{basePriority}\ \mathit{thread}?\})\} \\
\exists\, \mathit{priority} : \{\mathit{currentPriority}\ \mathit{thread}?\} \bullet \mathit{priorityQueues}' = \\
\qquad (\mathit{updatePriorityQueue}\ \mathit{priority}\ (\mathit{removeFromQueue}\ \mathit{thread}?) \fatsemi \\
\qquad \mathit{updatePriorityQueue}\ \mathit{priority}\ (\mathit{pushFront}\ \mathit{thread}?))\ \mathit{priorityQueues} \\
\mathit{priorityCeiling}' = \mathit{priorityCeiling} \land \mathit{interruptThreads}' = \mathit{interruptThreads} \\
\hline
\end{array}
$$

The second case, where the lock held has been taken more than once by the same thread, is described by the schema *PCEReleaseNestedLock*. This does not affect the components of *ThreadManager* or *ThreadInfo*, and takes as input the identifier of the object to be unlocked and the thread taking the lock, much like *PCEReleaseLock*. In this case, the object must be in the set of locks held by the thread and the *lockCount* value for the object is required to be greater than 1. The *lockCount* value for the object is decremented and the values for all other objects are unaffected. The other state components are unchanged. As in *PCERetakeLock*, the *current* thread is returned as *previous*!.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_ \mathit{PCEReleaseNestedLock} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{PCEScheduler} \\
\Xi \mathit{Scheduler} \\
\mathit{object}? : \mathit{ObjectID} \\
\mathit{thread}? : \mathit{ThreadID} \\
\mathit{previous}! : \mathit{ThreadID} \\
\hline
\mathit{object}? \in \mathit{locksHeld}\ \mathit{thread}? \\
\mathit{lockCount}\ \mathit{object}? > 1 \\
\mathit{lockCount}'\ \mathit{object}? = \mathit{lockCount}\ \mathit{object}? - 1 \\
\{\mathit{object}?\} \lhd \mathit{lockCount}' = \{\mathit{object}?\} \lhd \mathit{lockCount} \\
\mathit{lockHolder}' = \mathit{lockHolder} \land \mathit{locksHeld}' = \mathit{locksHeld} \\
\mathit{priorityCeiling}' = \mathit{priorityCeiling} \\
\mathit{previous}! = \mathit{current} \\
\hline
\end{array}
$$

The priority ceiling of an object can be set using the operation described by the schema *PCESetPriorityCeiling*, which takes an object identifier and a ceiling priority as input and does not affect the state of *Scheduler*. The object input must not have its lock held by any object, i.e. it must not be in the domain of *lockHolder*. The object is updated in the *priorityCeiling* map so that it maps to the given ceiling priority. The lock state is unaffected by this operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PCESetPriorityCeiling}} \\
\hline
\Delta \mathit{PCEScheduler} \\
\Xi \mathit{Scheduler} \\
\mathit{object}? : \mathit{ObjectID} \\
\mathit{ceiling}? : \mathit{Priority} \\
\hline
\mathit{object}? \notin \mathrm{dom}\, \mathit{lockHolder} \\
\mathit{priorityCeiling}' = \mathit{priorityCeiling} \oplus \{\mathit{object}? \mapsto \mathit{ceiling}?\} \\
\mathit{lockCount}' = \mathit{lockCount} \land \mathit{lockHolder}' = \mathit{lockHolder} \land \mathit{locksHeld}' = \mathit{locksHeld} \\
\hline
\end{array}
$$

In order to prevent deadlock, it is forbidden for a thread to suspend itself while holding a lock. To enforce this condition, we extend *ThreadSuspend* to the schema *PCESuspend*, which adds the precondition that *toSuspend* must hold no locks.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PCESuspend}} \\
\hline
\Delta \mathit{PCEScheduler} \\
\mathit{ThreadSuspend} \\
\hline
\mathit{locksHeld}\ \mathit{toSuspend}? = \varnothing \\
\hline
\end{array}
$$

Next, we specify interrupt handlers, which are similar to threads but require some extra handling in how they are started and finished.

3.4.2.4 Interrupt Handling

The SCJVM scheduler must manage interrupts, tracking the priority of each interrupt, the handler attached to it and the backing store to be used as the allocation context of the handler. The *PCEScheduler* state is extended to handle interrupts as specified in the *InterruptScheduler* schema. The state contains a function mapping each interrupt to a priority, which must be an interrupt priority. The state additionally contains functions mapping interrupts to their handlers, which are pairs of class and object identifiers representing interrupt handlers written as Java objects, the backing stores used as their allocation contexts, and the stacks used by the handlers. These functions are partial since not every interrupt will have a handler associated to it. There is also a set of identifiers of interrupts masked by currently running interrupts and a boolean flag that indicates if interrupts are enabled or not. Finally, since interrupts are managed as threads, there is a map, *interruptThreadMap*, from interrupt identifiers to threads running the interrupt handlers, which is partial because such a thread only exists for a running interrupt handler. The domains of the interrupt handler, allocation context and stack functions must be the same since they all apply only to interrupts with attached handlers. An interrupt that is running is always masked, so the domain of *interruptThreadMap* must be a subset of the masked thread set. The range of *interruptThreadMap* must also be the same as the set of interrupt threads defined earlier in the priority scheduler, since they are the threads used for interrupts. The base priority of each interrupt thread must be the same as the priority for the corresponding interrupt. This is specified by requiring that the composition of *interruptThreadMap* and the base priority map be a subset of the interrupt priority map so that an interrupt handler's actual base priority will match up with its stored priority.

$$
\begin{array}{|l}
\hline
\textit{InterruptScheduler} \\
\textit{PCEScheduler} \\
\textit{interruptPriority} : \textit{InterruptID} \to \textit{InterruptPriority} \\
\textit{interruptHandler} : \textit{InterruptID} \pfun \textit{ClassID} \times \textit{ObjectID} \\
\textit{interruptAC} : \textit{InterruptID} \pfun \textit{BackingStoreID} \\
\textit{interruptStack} : \textit{InterruptID} \pfun \textit{StackID} \\
\textit{maskedInterrupts} : \mathbb{P}\, \textit{InterruptID} \\
\textit{interruptsEnabled} : \mathbb{B} \\
\textit{interruptThreadMap} : \textit{InterruptID} \pfun \textit{ThreadID} \\
\hline
\mathrm{dom}\, \textit{interruptHandler} = \mathrm{dom}\, \textit{interruptAC} = \mathrm{dom}\, \textit{interruptStack} \\
\mathrm{dom}\, \textit{interruptThreadMap} \subseteq \textit{maskedInterrupts} \\
\mathrm{ran}\, \textit{interruptThreadMap} = \textit{interruptThreads} \\
\textit{interruptThreadMap} \fatsemi \textit{basePriority} \subseteq \textit{interruptPriority} \\
\hline
\end{array}
$$

Initialisation of *InterruptScheduler* occurs as specified in *InterruptSchedulerInit*, which is based on *PCESchedulerInit*. Initially, no interrupts have handlers attached, so the interrupt handler, allocation context and stack maps are empty. The *interruptThreadMap* and the *maskedInterrupt* set are also empty, since there are no interrupt handlers running. Interrupts are initially enabled, so *interruptsEnabled* is **True**. The interrupt priorities are unspecified as they are implementation-defined and cannot be changed by the user.

$$
\begin{array}{|l}
\hline
\textit{InterruptSchedulerInit} \\
\textit{InterruptScheduler}\,' \\
\textit{PCESchedulerInit} \\
\hline
\textit{interruptHandler}' = \varnothing \\
\textit{interruptAC}' = \varnothing \\
\textit{interruptStack}' = \varnothing \\
\textit{interruptThreadMap}' = \varnothing \\
\textit{maskedInterrupts}' = \varnothing \\
\textit{interruptsEnabled}' = \mathbf{True} \\
\hline
\end{array}
$$

Attaching a handler to a specified interrupt is provided for by the operation defined by the schema *InterruptAttachHandler*, which preserves the state of *PCEScheduler* and takes as input the identifier of the interrupt to attach the handler to along with the class, object, allocation context and stack of the handler to attach. The interrupt handler map is updated to associate the class and object with the specified interrupt. The interrupt allocation context map is similarly updated to associate the backing store given as allocation context to the interrupt, and the interrupt stack map is updated to associate the given stack identifier to the interrupt. The other state components that record information for interrupts are unaffected.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InterruptAttachHandler}} \\
\Delta \mathit{InterruptScheduler} \\
\Xi \mathit{PCEScheduler} \\
\mathit{interrupt}? : \mathit{InterruptID} \\
\mathit{handlerClass}? : \mathit{ClassID} \\
\mathit{handlerObject}? : \mathit{ObjectID} \\
\mathit{ac}? : \mathit{BackingStoreID} \\
\mathit{stack}? : \mathit{StackID} \\
\hline
\mathit{interruptHandler}' = \\
\qquad \mathit{interruptHandler} \oplus \{\mathit{interrupt}? \mapsto (\mathit{handlerClass}?, \mathit{handlerObject}?)\} \\
\mathit{interruptAC}' = \mathit{interruptAC} \oplus \{\mathit{interrupt}? \mapsto \mathit{ac}?\} \\
\mathit{interruptStack}' = \mathit{interruptStack} \oplus \{\mathit{interrupt}? \mapsto \mathit{stack}?\} \\
\mathit{interruptPriority}' = \mathit{interruptPriority} \\
\mathit{maskedInterrupts}' = \mathit{maskedInterrupts} \\
\mathit{interruptThreadMap}' = \mathit{interruptThreadMap} \\
\mathit{interruptsEnabled}' = \mathit{interruptsEnabled} \\
\hline
\end{array}$$

Detaching an interrupt handler is defined by *InterruptDetachHandler*, which takes a interrupt identifier as input and removes it from the *interruptHandler*, *interruptAC* and *interruptStack* maps. We omit its definition here as it is similar to *InterruptAttachHandler*.

Getting the priority of a given interrupt is described by the *InterruptGetPriority* schema. We omit it here since it is a simple operation that just applies *interruptPriority* to its input.

The operations of enabling and disabling interrupts simply set the value of the boolean flag indicating whether or not interrupts are enabled. Because these operations only affect one state component, we define a schema *InterruptEnableFixedVars* to more briefly state that all other state components remain the same.

$$\mathit{InterruptEnableFixedVars} == \Xi \mathit{InterruptScheduler} \setminus (\mathit{interruptsEnabled})$$

As the operations of enabling and disabling interrupts are similar, we just present the schema *InterruptEnable* here, omitting the *InterruptDisable* schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InterruptEnable}} \\
\Delta \mathit{InterruptScheduler} \\
\mathit{InterruptEnableFixedVars} \\
\hline
\mathit{interruptsEnabled}' = \mathit{True} \\
\hline
\end{array}$$

When a lock is taken and the priority of the thread taking the lock is raised to the priority ceiling of the locked object, it may be raised to an interrupt priority, and that prevents interrupts of lower priority from executing. However, the execution of interrupts is usually handled by hardware and so interrupts cannot be scheduled as ordinary threads, although they are modelled as such here in order to specify their relationship to the non-interrupt threads. The interrupts prevented from executing by a lock must instead be masked to prevent them firing in hardware. We specify this by extending *PCETakeLock* and *PCEReleaseLock* to change the *maskedInterrupts* set.

The first of these extended operations is *InterruptTakeLock*, which behaves as *PCETakeLock*, but also changes *maskedInterrupts*. The *maskedInterrupts* set is defined to include all threads with a lower priority than the priority of any interrupt already running, or the priority ceiling of any object that has been locked. The other state components are not affected.

$$
\begin{array}{|l}
\textit{InterruptTakeLock} \\
\hline
\Delta \textit{InterruptScheduler} \\
\textit{PCETakeLock} \\
\hline
\textit{maskedInterrupts}' = \{\, i : \textit{InterruptID} \mid \\
\quad (\exists\, j : \operatorname{dom} \textit{interruptThreadMap}' \bullet \textit{interruptPriority}\ i \leq \textit{interruptPriority}\ j) \\
\quad \lor (\exists\, \textit{obj} : \operatorname{dom} \textit{lockHolder}' \bullet \textit{interruptPriority}\ i \leq \textit{priorityCeiling}\ \textit{obj})\} \\
\textit{interruptPriority}' = \textit{interruptPriority} \land \textit{interruptHandler}' = \textit{interruptHandler} \\
\textit{interruptAC}' = \textit{interruptAC} \land \textit{interruptStack}' = \textit{interruptStack} \\
\hline
\end{array}
$$

We also define an operation *InterruptReleaseLock*, which behaves as *PCEReleaseLock* and updates *maskedInterrupts* in the same way as *InterruptTakeLock*. We omit it here due to its similarity with *InterruptTakeLock*. Note that similar extensions are not required for the *PCERetakeLock* and *PCEReleaseNestedLock* cases, since they do not change the locks held.

When an interrupt is fired, it is handled as described by *HandleInterrupt*, defined below. The identifier of the interrupt to be handled is passed as an input to the operation, and the handler thread identifier, allocation context and stack are output so that they can be communicated to the core execution environment and memory manager. A boolean is also output to indicate whether or not the interrupt was actually handled. For the interrupt to be handled, interrupts must be enabled, the interrupt must not be masked, and the interrupt must have a handler attached. The new interrupt handler thread's identifier is chosen from the *free* thread identifiers. That identifier is removed from the *free* thread identifiers, and added to the *started* thread identifiers, which are then further updated by the choosing of a new *current* thread as specified by *PickNewCurrent*. The current and base priority of the new thread are set to the given interrupt's priority and the thread's identifier is placed at the back of the queue for its priority. The set of interrupt threads is updated to include the identifier of the new handler thread and all threads with priority less than or equal to the interrupt's priority are added to the set of masked interrupts.Note that locks do not need to be considered in this update of *maskedInterrupts*. Any lock that would cause additional interrupts to be included would also prevent *interrupt*? from firing, since it would already be masked. The other state components are unaffected, though the thread class, method and argument maps must be updated to include the new thread. It does not matter what values are included for the new thread, but the other values in the maps are required to remain the same. The values in the allocation context and stack maps for the interrupt are output, and, since the interrupt was handled, the boolean flag output is true.

HandleInterrupt

$$\begin{array}{l}
\Delta InterruptScheduler \\
ChangeFreeStartedCurrent \\
interrupt? : InterruptID \\
handler! : ThreadID \\
ac! : BackingStoreID \\
stack! : StackID \\
handled! : \mathbb{B} \\
previous! : ThreadID \\
\hline
interruptsEnabled = True \\
interrupt? \notin maskedInterrupts \\
interrupt? \in \operatorname{dom} interruptHandler \\
handler! \in free \\
free' = free \setminus \{handler!\} \\
\exists\, newStarted? : \{started \cup \{handler!\}\} \bullet PickNewCurrent \\
currentPriority' = currentPriority \oplus \{handler! \mapsto interruptPriority\ interrupt?\} \\
basePriority' = basePriority \oplus \{handler! \mapsto interruptPriority\ interrupt?\} \\
\exists\, priority : \{interruptPriority\ interrupt?\} \bullet \\
\quad priorityQueues' = \\
\qquad updatePriorityQueue\ priority\ (pushBack\ handler!)\ priorityQueues \\
interruptThreads' = interruptThreads \cup \{handler!\} \\
maskedInterrupts' = \\
\quad \{\, i : InterruptID \mid interruptPriority\ i \leq interruptPriority\ interrupt?\, \} \\
\{handler!\} \mathbin{\lhd\!\!\!-} threadClass' = \{handler!\} \mathbin{\lhd\!\!\!-} threadClass \\
\{handler!\} \mathbin{\lhd\!\!\!-} threadMethod' = \{handler!\} \mathbin{\lhd\!\!\!-} threadMethod \\
\{handler!\} \mathbin{\lhd\!\!\!-} threadArgs' = \{handler!\} \mathbin{\lhd\!\!\!-} threadArgs \\
interruptPriority' = interruptPriority \\
priorityCeiling' = priorityCeiling \land lockHolder' = lockHolder \\
lockCount' = lockCount \land locksHeld' = locksHeld \\
interruptHandler' = interruptHandler \land interruptAC' = interruptAC \\
ac! = interruptAC\ interrupt? \\
stack! = interruptStack\ interrupt? \\
handled! = True
\end{array}$$

If interrupts are disabled, the interrupt is masked, or the interrupt has no handler attached then it is silently ignored. This is described in the schema *IgnoreInterrupt*, which does not change the state. The interrupt's identifier is taken as an input and a boolean is output to indicate that the interrupt was not handled. The identifier of the *current* thread is output as *previous*! to match the interface of *HandleInterrupt*, since the *current* thread is not updated by *PickNewCurrent*.

$$
\begin{array}{l}
\textit{IgnoreInterrupt} \\
\hline
\Xi \mathit{InterruptScheduler} \\
\mathit{interrupt}? : \mathit{InterruptID} \\
\mathit{handled}! : \mathbb{B} \\
\mathit{previous}! : \mathit{ThreadID} \\
\hline
\mathit{interruptsEnabled} = \mathit{False} \lor \\
\quad \mathit{interrupt}? \in \mathit{maskedInterrupts} \lor \\
\quad \mathit{interrupt}? \notin \operatorname{dom} \mathit{interruptHandler} \\
\mathit{handled}! = \mathit{False} \\
\mathit{previous}! = \mathit{current} \\
\hline
\end{array}
$$

When an interrupt handler ends, the interrupt handler thread is destroyed, removing the information about it from the scheduler as described by the schema *InterruptEnd*, which takes a thread identifier, *thread*?, as input. The thread must be an interrupt thread for this operation to be used, since it represents the remaining case to be specified when a thread ends (in addition to *ThreadDestroy*). The information about the thread is removed from the *ThreadInfo* maps and the thread's identifier is filtered out of the queue for its priority. The queues for the other priorities are unaffected. A new *current* thread is then chosen as specified in *PickNewCurrent*, with the thread's identifier removed from *started*, as in *ThreadDestroy*, and the old *current* thread is output as *previous*!. The thread is removed from the interrupt threads set and the masked interrupts are updated as in *InterruptTakeLock* and *InterruptReleaseLock*. The other state components are unaffected.

$$
\begin{array}{l}
\textit{InterruptEnd} \\
\hline
\Delta \mathit{InterruptScheduler} \\
\mathit{RemoveThreadInfo} \\
\mathit{thread}? : \mathit{ThreadID} \\
\mathit{previous}! : \mathit{ThreadID} \\
\hline
\mathit{thread}? \in \mathit{interruptThreads} \\
\exists\, \mathit{priority} : \{\mathit{currentPriority}\ \mathit{thread}?\} \bullet \mathit{priorityQueues}' = \\
\quad \mathit{updatePriorityQueue}\ \mathit{priority}\ (\mathit{removeFromQueue}\ \mathit{current})\ \mathit{priorityQueues} \\
\mathit{free}' = \mathit{free} \cup \{\mathit{thread}?\} \\
\mathit{interruptThreads}' = \mathit{interruptThreads} \setminus \{\mathit{thread}?\} \\
\mathit{maskedInterrupts}' = \{\, i : \mathit{InterruptID} \mid \\
\quad (\exists\, j : \operatorname{dom} \mathit{interruptThreadMap}' \bullet \mathit{interruptPriority}\ i \leq \mathit{interruptPriority}\ j) \\
\quad \lor (\exists\, \mathit{obj} : \operatorname{dom} \mathit{lockHolder} \bullet \mathit{interruptPriority}\ i \leq \mathit{priorityCeiling}\ \mathit{obj})\} \\
\exists\, \mathit{newStarted}? : \{\mathit{started} \setminus \{\mathit{thread}?\}\};\ \mathit{started}0 : \mathbb{F}\ \mathit{ThreadID} \bullet \\
\quad \mathit{PickNewCurrent}[\mathit{started}0/\mathit{started}'] \land \mathit{started}' = \mathit{started}0 \setminus \{\mathit{thread}?\} \\
\mathit{interruptPriority}' = \mathit{interruptPriority} \\
\mathit{priorityCeiling}' = \mathit{priorityCeiling} \land \mathit{lockHolder}' = \mathit{lockHolder} \\
\mathit{lockCount}' = \mathit{lockCount} \land \mathit{locksHeld}' = \mathit{locksHeld} \\
\mathit{interruptHandler}' = \mathit{interruptHandler} \land \mathit{interruptAC}' = \mathit{interruptAC} \\
\mathit{created}' = \mathit{created} \land \mathit{blocked}' = \mathit{blocked} \\
\hline
\end{array}
$$

InterruptEnd is used along with *ThreadDestroy* to specify the `endThread` operation. Thus, we also lift *RThreadDestroy* to act over the *InterruptScheduler* so that it can be combined with *InterruptEnd*. We call this lifted version *RInterruptThreadDestroy*. It leaves the components of *InterruptScheduler* that are not specified by *RThreadDestroy* unchanged.

The schemas declared so far are also lifted to robust actions. Although the schemas we have defined so far specify all the operations of the scheduler, the core execution environment may not always be ready to accept a thread switch, so we place them in a queue until the core execution environment is ready to accept them. We specify this in the next section.

3.4.2.5 Thread Switches

We define a datatype *ThreadSwitchInfo*, which contains the information for a thread switch (a pair giving the thread switched from and to). This type also contains a variant specifying the information for a thread start, since both thread starts and switches are communicated to the core execution environment so they must be queued together.

$$
\begin{array}{l}
\mathit{ThreadSwitchInfo} ::= \\
\quad \mathit{switch}\langle\!\langle \mathit{ThreadID} \times \mathit{ThreadID} \rangle\!\rangle \\
\quad \mid \mathit{start}\langle\!\langle \mathit{ThreadID} \times \mathit{BackingStoreID} \times \mathit{StackID} \times \mathit{ClassID} \times \mathit{MethodID} \times \mathrm{seq}\ \mathit{Word} \rangle\!\rangle
\end{array}
$$

The state of the queue of thread switches and starts is specified in the schema *SwitchManager*. The queue itself is the state component *switchQueue*, which is a sequence of *ThreadSwitchInfo*. In addition, this behaviour of queueing thread switches may cause the *current* thread in the scheduler to differ from the thread that is running in the core execution environment. Since some of the scheduler operations, such as suspending a thread, are intended to operate only on the current thread (from the perspective of the core execution environment, which uses the operations), we track this current thread in *SwitchManager* as *phantomCurrent*. Since unnecessary thread switches are undesirable (as they slow down execution), the invariant of *SwitchManager* also specifies that *switchQueue* must not contain a switch from a thread to itself nor a switch back to a thread immediately after a switch from it.

$$
\begin{array}{|l}
\textit{SwitchManager} \\
\hline
\mathit{switchQueue} : \mathrm{seq}\ \mathit{ThreadSwitchInfo} \\
\mathit{phantomCurrent} : \mathit{ThreadID} \\
\hline
\forall t : \mathit{ThreadID} \bullet \mathit{switch}\,(t, t) \notin \mathrm{ran}\,\mathit{switchQueue} \\
\neg\ \exists t1, t2 : \mathit{ThreadID} \bullet \exists u, v : \mathrm{seq}\ \mathit{ThreadSwitchInfo} \bullet \\
\qquad u \frown \langle \mathit{switch}\,(t1, t2), \mathit{switch}\,(t2, t1) \rangle \frown v = \mathit{switchQueue} \\
\hline
\end{array}
$$

Initially, the *switchQueue* is empty, and *phantomCurrent* is set to *main*, since that is the thread that is running when the SCJVM starts. This is specified in *SwitchManagerInit*.

$$
\begin{array}{|l}
\textit{SwitchManagerInit} \\
\hline
\mathit{SwitchManager}\,' \\
\hline
\mathit{switchQueue}' = \varnothing \\
\mathit{phantomCurrent}' = \mathit{main} \\
\hline
\end{array}
$$

When thread switches are pushed to the thread switch queue, there are three cases, to ensure that unnecessary thread switches are edited out of the queue to preserve the invariants mentioned above. The first case, described by *PushThreadSwitchNormal*, applies when the thread switch does not need editing out of the queue. It takes as input the identifiers of the threads switched from, *fromThr*?, and to, *toThr*?, as do all of the cases for pushing thread switches. It requires that *fromThr*? and *toThr*? are not the same, and that the back of the queue is not a switch from *toThr*? back to *fromThr*?. The new switch is pushed to the back of *switchQueue* and *phantomCurrent* is unchanged, since the thread switch has not yet been passed to the core execution environment.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PushThreadSwitchNormal}} \\
\hline
\Delta \mathit{SwitchManager} \\
\mathit{fromThr}?, \mathit{toThr}? : \mathit{ThreadID} \\
\hline
\mathit{fromThr}? \neq \mathit{toThr}? \land \neg \langle \mathit{switch}\,(\mathit{toThr}?, \mathit{fromThr}?) \rangle \ \mathit{suffix}\ \mathit{switchQueue} \\
\mathit{switchQueue}' = \mathit{switchQueue} \frown \langle \mathit{switch}\,(\mathit{fromThr}?, \mathit{toThr}?) \rangle \\
\mathit{phantomCurrent}' = \mathit{phantomCurrent} \\
\hline
\end{array}$$

In the second case, described by *PushThreadSwitchSelf*, *fromThr*? is the same as *toThr*?. In this case, the state of *SwitchManager* is unchanged, since a switch from a thread to itself is unnecessary, so it is not pushed.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PushThreadSwitchSelf}} \\
\hline
\Xi \mathit{SwitchManager} \\
\mathit{fromThr}?, \mathit{toThr}? : \mathit{ThreadID} \\
\hline
\mathit{fromThr}? = \mathit{toThr}? \\
\hline
\end{array}$$

In the third case, described by *PushThreadSwitchReverse*, there is a switch from *toThr*? to *fromThr*? at the back of the *switchQueue*. Since the switch that would be added to *switchQueue* reverses that switch, we remove that switch from the back of *switchQueue* rather than adding the new switch. The *phantomCurrent* is unchanged.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PushThreadSwitchReverse}} \\
\hline
\Delta \mathit{SwitchManager} \\
\mathit{fromThr}?, \mathit{toThr}? : \mathit{ThreadID} \\
\hline
\langle \mathit{switch}\,(\mathit{toThr}?, \mathit{fromThr}?) \rangle \ \mathit{suffix}\ \mathit{switchQueue} \\
\mathit{switchQueue}' = \mathit{front}\ \mathit{switchQueue} \\
\mathit{phantomCurrent}' = \mathit{phantomCurrent} \\
\hline
\end{array}$$

The overall specification of pushing a new thread switch to the queue is given by the disjunction of these three schemas.

$$\begin{array}{l}
\mathit{PushThreadSwitch} == \\
\quad \mathit{PushThreadSwitchNormal} \lor \mathit{PushThreadSwitchSelf} \lor \mathit{PushThreadSwitchReverse}
\end{array}$$

Pushing a thread start to the *switchQueue* is described by *PushThreadStart*. It takes as input the information required for the thread start: the identifier of the thread, the backing store for the thread, the stack for the thread, the class and method identifier of the method to be executed on the thread, and the list of arguments to the method. The thread start is placed at the back of the *switchQueue* and *phantomCurrent* is unaffected.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{PushThreadStart} \quad}} \\
\Delta \mathit{SwitchManager} \\
\mathit{thread}? : \mathit{ThreadID} \\
\mathit{bsid}? : \mathit{BackingStoreID} \\
\mathit{stack}? : \mathit{StackID} \\
\mathit{class}? : \mathit{ClassID} \\
\mathit{method}? : \mathit{MethodID} \\
\mathit{args}? : \mathrm{seq}\, \mathit{Word} \\
\hline
\mathit{switchQueue}' = \\
\qquad \mathit{switchQueue} \frown \langle \mathit{start}\, (\mathit{thread}?, \mathit{bsid}?, \mathit{stack}?, \mathit{class}?, \mathit{method}?, \mathit{args}?) \rangle \\
\mathit{phantomCurrent}' = \mathit{phantomCurrent} \\
\hline
\end{array}
$$

A thread switch is popped from *switchQueue* as described in *PopThreadSwitch*. It requires that the *switchQueue* be non-empty, and that its front element is a thread switch, rather than a thread start. The identifiers of the threads given in the thread switch at the front of *switchQueue* are output as $\mathit{fromThr}!$ and $\mathit{toThr}!$. The element at the front of *switchQueue* is removed and *phantomCurrent* is set to $\mathit{toThr}!$, since that is the thread running in the core execution environment after the switch has been accepted.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{PopThreadSwitch} \quad}} \\
\Delta \mathit{SwitchManager} \\
\mathit{fromThr}!, \mathit{toThr}! : \mathit{ThreadID} \\
\hline
\mathit{switchQueue} \neq \varnothing \land \mathit{head}\, \mathit{switchQueue} \in \mathrm{ran}\, \mathit{switch} \\
\mathit{fromThr}! = ((\mathit{switch}^{\sim})\, (\mathit{head}\, \mathit{switchQueue})).1 \\
\mathit{toThr}! = ((\mathit{switch}^{\sim})\, (\mathit{head}\, \mathit{switchQueue})).2 \\
\mathit{switchQueue}' = \mathit{tail}\, \mathit{switchQueue} \\
\mathit{phantomCurrent}' = \mathit{toThr}! \\
\hline
\end{array}
$$

PopThreadStart describes popping a thread start from *switchQueue*. It requires that the *switchQueue* be non-empty and that the element at the front of the *switchQueue* is a thread start. The information for the thread start is output as $\mathit{threadStartInfo}!$, and it is removed from the front of *switchQueue*. The *phantomCurrent* is unaffected in this case.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{PopThreadStart} \quad}} \\
\Delta \mathit{SwitchManager} \\
\mathit{threadStartInfo}! \\
\quad : \mathit{ThreadID} \times \mathit{BackingStoreID} \times \mathit{StackID} \times \mathit{ClassID} \times \mathit{MethodID} \times \mathrm{seq}\, \mathit{Word} \\
\hline
\mathit{switchQueue} \neq \varnothing \land \mathit{head}\, \mathit{switchQueue} \in \mathrm{ran}\, \mathit{start} \\
\mathit{threadStartInfo}! = (\mathit{start}^{\sim})\, (\mathit{head}\, \mathit{switchQueue}) \\
\mathit{switchQueue}' = \mathit{tail}\, \mathit{switchQueue} \\
\mathit{phantomCurrent}' = \mathit{phantomCurrent} \\
\hline
\end{array}
$$

This concludes the Z portion of the scheduler model. The operations must now be lifted to *Circus* actions accessed via the channels declared earlier and the interaction with interrupt signals must be specified.

3.4.2.6 Scheduler Operations

The scheduler model offers the services detailed in this section. The operations described in Section 3.2 are all implemented here. Some additional actions are also defined for handling interrupts, and communicating thread starts and switches to the core execution environment.

The state of the scheduler process is the conjunction of *InterruptScheduler*, which contains the state of the priority ceiling emulation manager and priority scheduler as well as the interrupt manager state, and *SwitchManager*.

$$\mathbf{state}\ \mathit{InterruptScheduler} \land \mathit{SwitchManager}$$

The scheduler is initialised with the main thread's priority via the *Sinit* channel and the initial state is as described by *InterruptSchedulerInit* and *SwitchManagerInit*.

$$\mathit{Init} \mathrel{\widehat=} \mathit{Sinit}?\mathit{mainPriority} \longrightarrow \big(\mathit{InterruptSchedulerInit} \land \mathit{SwitchManagerInit}\big)$$

The services that output constant priority values, such as `getMaxSoftwarePriority`, simply output the relevant value over their channel and then output a report of *Sokay*, as shown in the example of the *GetMaxSoftwarePriority* action below.

$$\begin{aligned}&\mathit{GetMaxSoftwarePriority} \mathrel{\widehat=}\\ &\quad \mathit{SgetMaxSoftwarePriority}!\mathit{maxSwPriority} \longrightarrow \mathit{Sreport}!\mathit{Sokay} \longrightarrow \mathbf{Skip}\end{aligned}$$

The actions for getting the main and current threads are specified in a similar way, since they just output the *main* and *current* thread identifiers respectively.

The other operations are lifted to *Circus* actions from the Z schemas defined earlier. The correspondence between the services described in section Section 3.2, the *Circus* actions described here, and the Z schemas defined earlier is shown in Table 3.5. These actions follow a common pattern, seen in the *GetInterruptPriority* action below, which is similar to the lifting of the memory manager operations in Section 3.4.1. The signal to perform the operation, along with the inputs to the operation, is communicated via the operation's channel. The operation is then performed as specified by the corresponding Z schema. Any outputs are communicated on a return channel and the error report from the operation is sent on the *Sreport* channel.

$$\begin{aligned}&\mathit{GetInterruptPriority} \mathrel{\widehat=} \mathbf{var}\ \mathit{priority} : \mathit{Priority};\ \mathit{report} : \mathit{SReport} \bullet\\ &\quad \mathit{SgetInterruptPriority}?\mathit{interrupt} \longrightarrow \big(\mathit{RInterruptGetPriority}\big);\\ &\quad \mathit{SgetInterruptPriorityRet}!\mathit{priority} \longrightarrow \mathit{Sreport}!\mathit{report} \longrightarrow \mathbf{Skip}\end{aligned}$$

Many of the actions, however, deviate from this pattern since the scheduler must respond to external events and communicate with the core execution environment.

Many scheduler operations change the *current* thread, which requires a thread switch to be pushed to *switchQueue*, for communication to the core execution environment. However, a thread switch should only be pushed if the operation completes successfully. We thus specify the pushing of a thread switch in a separate action, *HandleThreadSwitch*, which is parametrised by an *SReport* value and identifiers of the threads switched from and to. If the *SReport* value is *Sokay*, then the thread switch is pushed as specified by *PushThreadSwitch*. Otherwise, the action terminates without changing the state.

$$\begin{aligned}&\mathit{HandleThreadSwitch} \mathrel{\widehat=} \mathbf{val}\ \mathit{report} : \mathit{SReport};\ \mathbf{val}\ \mathit{fromThr}, \mathit{toThr} : \mathit{ThreadID} \bullet\\ &\quad \mathbf{if}\ \mathit{report} = \mathit{Sokay} \longrightarrow \big(\mathit{PushThreadSwitch}\big)\\ &\quad \ [\!]\ \mathit{report} \neq \mathit{Sokay} \longrightarrow \mathit{Skip}\\ &\quad \mathbf{fi}\end{aligned}$$

Service	*Circus* action	Z schema
`getMaxSoftwarePriority`	$GetMaxSoftwarePriority$	(none)
`getMinSoftwarePriority`	$GetMinSoftwarePriority$	(none)
`getNormSoftwarePriority`	$GetNormSoftwarePriority$	(none)
`getMaxHardwarePriority`	$GetMaxHardwarePriority$	(none)
`getMinHardwarePriority`	$GetMinHardwarePriority$	(none)
`getMainThread`	$GetMainThread$	(none)
`makeThread`	$MakeThread$	$RThreadCreate$
`startThread`	$StartThread$	$RThreadStart$
`getCurrentThread`	$GetCurrentThread$	(none)
`suspendThread`	$SuspendThread$	$RPCESuspend$
`resumeThread`	$ResumeThread$	$RThreadResume$
`setPriorityCeiling`	$SetPriorityCeiling$	$RPCESetPriorityCeiling$
`takeLock`	$TakeLock$	$RInterruptTakeLock$
`releaseLock`	$ReleaseLock$	$RInterruptReleaseLock$
`attachInterruptHandler`	$AttachInterruptHandler$	$RInterruptAttachHandler$
`detachInterruptHandler`	$DetachInterruptHandler$	$RInterruptDetachHandler$
`getInterruptPriority`	$GetInterruptPriority$	$RInterruptGetPriority$
`disableInterrupts`	$DisableInterrupts$	$RInterruptDisable$
`enableInterrupts`	$EnableInterrupts$	$RInterruptEnable$
`endThread`	$EndThread$	$RInterruptEnd \lor RInterruptThreadDestroy$

Table 3.5: The relationship between the scheduler services and the *Circus* actions and Z schemas defining them

An example of an action that makes use of *HandleThreadSwitch* is *ResumeThread*, shown below. This action follows much the same format as that shown for *GetInterruptPriority* above, but ends with the *HandleThreadSwitch* action, passing in the *report* value output by the operation, along with the *previous* thread identifier output as the thread switched from and the new *current* thread as the thread switched to.

$$\begin{array}{l} ResumeThread \mathrel{\widehat=} \mathbf{var}\, thread, previous : ThreadID;\ report : SReport \bullet \\ \quad SresumeThread?thread \longrightarrow \big(RThreadResume\big); \\ \quad Sreport!report \longrightarrow HandleThreadSwitch(report, previous, current) \end{array}$$

The other actions followed by *HandleThreadSwitch* are *TakeLock*, *ReleaseLock*, *SuspendThread*, and *EndThread*. These are all operations that act upon the current thread, so *phantomCurrent* is passed to schemas defining them as the thread they are to act upon, since that is the thread that is running in the core execution environment when the scheduler operation is used. This can be seen in the *TakeLock* action, which handles the current thread taking the lock on an object, where *phantomCurrent* is passed to *RInterruptTakeLock*.

$$\begin{array}{l} TakeLock \mathrel{\widehat=} \mathbf{var}\, report : SReport;\ previous : ThreadID \bullet \\ \quad StakeLock?object \longrightarrow \big(RInterruptTakeLock[phantomCurrent/thread?]\big); \\ \quad Sreport!report \longrightarrow HandleThreadSwitch(report, previous, current) \end{array}$$

The *StartThreads* action requires some additional communication with the core execution environment and must loop over all of the threads in the input set. The input set *threadsInfo*, containing triples of thread, backing store and stack identifiers, is received via the *SstartThreads* channel. The set of thread identifiers in this input set is used to update the scheduler's state as described in *RThreadStarts*. If there is no error report from *RThreadStarts*, then, using replicated sequential composition over the pairs in *threadsInfo*, the thread identifier, *thread*, the backing store identifier, *bsid*, and the stack identifier, *stack*, are extracted, and the thread start is pushed to the *switchQueue* as specified in *PushThreadStart*. A backing store is only input to the scheduler in this action, not when the thread is created, since SCJ requires the memory areas for threads to be created as the threads are started. After all the thread starts have been pushed in sequence, the *report* is output over the *Sreport* channel and a thread switch is pushed as in the *HandleThreadSwitch* action. Pushing the thread switch only after pushing all the thread starts ensures that the threads are started at the same time, before any of them are switched to.

```
$StartThreads \mathrel{\widehat=}$
    $\mathbf{var}\ threadsInfo : \mathbb{F}(ThreadID \times BackingStoreID \times StackID) \bullet$
    $\mathbf{var}\ report : SReport;\ previous : ThreadID \bullet$
    $SstartThreads?ts \longrightarrow threadsInfo := ts;$
    $\big(\exists\, toStart? == \{t : threadsInfo \bullet t.1\} \bullet RThreadStarts\big);$
    $\mathbf{if}\ report = Sokay \longrightarrow (\mathop{;}\ threadinfo : threadsInfo \bullet$
        $\mathbf{var}\ thread : ThreadID;\ bsid : BackingStoreID;\ stack : StackID \bullet$
        $thread, bsid, stack := threadinfo.1, threadinfo.2, threadinfo.3;$
        $\mathbf{var}\ class : ClassID;\ method : MethodID;\ args : \mathrm{seq}\ Word \bullet$
        $class, method, args :=$
            $threadClass\ thread, threadMethod\ thread, threadArgs\ thread;$
        $\big(PushThreadStart\big))$
    $[\!]\ report \neq Sokay \longrightarrow \mathbf{Skip}$
    $\mathbf{fi};$
    $Sreport!report \longrightarrow HandleThreadSwitch(report, previous, current)$
```

The action of enabling interrupts must signal the hardware using the *HWenableInterrupts* channel in addition to updating the scheduler's state as described in *RInterruptEnable*.

```
$EnableInterrupts \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet$
    $SenableInterrupts \longrightarrow \big(RInterruptEnable\big);$
    $HWenableInterrupts \longrightarrow Sreport!report \longrightarrow \mathbf{Skip}$
```

Disabling of interrupts is similar, using the *HWdisableInterrupts* channel.

Interrupt handling must be done in response to a signal from hardware, so it is a separate action although it is not one of the public SCJVM services. An interrupt is handled by calling the `handle()` method of the interrupt handler object, which is represented by a method identifier *handleID* in our model.

$$| \quad handleID : MethodID$$

We define the handling of a given interrupt as a *Circus* action, *Handle*, which takes the identifier of the interrupt as a parameter. The interrupt handling specification is split into two cases: the case where the interrupt is actually handled, as described in *HandleInterrupt*, and the case where the interrupt is ignored, as described in *IgnoreInterrupt*. The Z schemas to handle each case are placed in disjunction and a *Circus* if statement is used to check the boolean output in order to determine which case took effect. If the interrupt was handled, the instruction to start the interrupt handler's thread is pushed to *switchQueue* as specified by *PushThreadStart*. The allocation context and class passed to the core execution environment are those given when the handler was attached to the interrupt. The method identifier used is *handleID* and the object identifier of the handler object is given as the only argument of the method. An instruction to switch to a new thread is then pushed to *switchQueue* via *PushThreadSwitch*, with *previous* as the thread switched from and *current* as the thread switched to. In the case where the interrupt was ignored, the action simply terminates.

```
$Handle \mathrel{\widehat=} \mathbf{val}\, interrupt : InterruptID \bullet$
    $\mathbf{var}\, handler, previous : ThreadID \bullet$
    $\mathbf{var}\, ac : BackingStoreID;\ stack : StackID;\ handled : \mathbb{B} \bullet$
    $\big(HandleInterrupt \lor IgnoreInterrupt\big);$
    $\mathbf{if}\, handled = True \longrightarrow$
        $\mathbf{var}\, class : ClassID;\ method : MethodID;\ args : \operatorname{seq} Word \bullet$
        $class, method, args :=$
            $(interruptHandler\ interrupt).1, handleID, \langle (interruptHandler\ interrupt).2\rangle;$
        $\big(PushThreadStart[handler/thread?, ac/bsid?]\big);$
        $\big(PushThreadSwitch[previous/fromThr?, current/toThr?]\big)$
    $[\!]\ handled = False \longrightarrow \mathbf{Skip}$
    $\mathbf{fi}$
```

There are two signals that cause the scheduler to handle an interrupt. The first is an interrupt coming from hardware via the *HWinterrupt* channel. We use input prefixing to require that this interrupt not be the clock interrupt since the clock interrupt is handled by the real-time clock. The interrupt is handled as described by the *Handle* action above.

$$\begin{aligned}&HandleNonclockInterrupt \mathrel{\widehat=}\\&\quad HWinterrupt?interrupt : (interrupt \neq clockInterrupt) \longrightarrow Handle(interrupt)\end{aligned}$$

The second signal that causes interrupt handling is the clock interrupt forwarded from the real-time clock via the *RTCclockInterrupt* channel. This is handled as if it had the identifier of the clock interrupt.

$$HandleClockInterrupt \mathrel{\widehat=} RTCclockInterrupt \longrightarrow Handle(clockInterrupt)$$

The final types of communications that the scheduler must handle are communications with the core execution environment to signal thread switches and starts when it is ready to perform them. Signalling a thread switch is performed by the *SwitchThread* action. This action is guarded by the precondition of *PopThreadSwitch*, so that it only offers to communicate with the core execution environment if there is a thread switch at the front of the *switchQueue*. When the action occurs, the thread switch is popped from the *switchQueue* via *PopThreadSwitch*, and the thread switch is communicated to the core execution environment via the *CEEswitchThread* channel. Note that, due to the way communications interact with external choice in *Circus*, the state is not updated as described by the *PopThreadSwitch* schema unless the communication occurs.

$$\begin{aligned}&SwitchThread \mathrel{\widehat=} \mathbf{var}\, fromThr, toThr : ThreadID \bullet\\&\quad (\mathbf{pre}\, PopThreadSwitch)\ \&\ \big(PopThreadSwitch\big);\\&\quad CEEswitchThread!fromThr!toThr \longrightarrow \mathbf{Skip}\end{aligned}$$

The action for popping a thread start from *switchQueue* is similar to *SwitchThread*, but uses *PopThreadStart* and communicates on the *CEEstartThread* channel. When *switchQueue* is empty, the scheduler offers communication on the *CEEproceed* channel to indicate that there are no pending thread starts or switches, and so the core execution environment can proceed with execution. This is specified in the *Proceed* action.

$$Proceed \mathrel{\widehat=} (switchQueue = \varnothing) \mathbin{\&} CEEproceed!current \longrightarrow \mathbf{Skip}$$

The scheduler continuously presents all its operations in a loop. Any operation can be chosen once the previous operation has completed.

$$Loop \mathrel{\widehat=} GetMainThread \mathbin{\Box} MakeThread \mathbin{\Box} StartThreads \cdots \mathbin{;} Loop$$

The main action of the scheduler process first requires initialisation and then enters the operation loop declared above.

$\bullet\ Init \mathbin{;} Loop$

end

This concludes the specification of the scheduler. We have specified threads and information about them, including their priority, whether they are available to run or not, and the method information required to begin execution of the thread. We specified the priority scheduler, which sorts the executable threads into queues by priority and selects the thread at the front of the highest non-empty priority queue to run. This includes the operations to create, start, destroy, suspend and resume threads. A mechanism for locking objects to prevent interference has also been specified, with priority ceiling emulation as a mechanism for avoiding priority inversion problems. We have also described the mechanism by which interrupt handlers are specified and how interrupt processing is performed by starting interrupt threads. Finally, we have lifted the scheduler operations to *Circus* actions accessed via channels and specified the relation of the scheduler to the hardware, memory manager and core execution environment.

3.4.3 Real-time Clock

The SCJVM real-time clock provides an interface to a hardware real-time clock, which is used by the SCJ clock API. The periodic clock interrupt from the hardware is handled by the SCJVM clock and used to manage alarms that trigger when a certain time is reached. If an alarm is set, the interrupt is passed to the scheduler when the alarm triggers; the SCJ API implementation should attach an interrupt handler to it that simply calls the `triggerAlarm()` method of `Clock` for the real-time clock.

The type used for interrupt identifiers is the same as that used by the scheduler. We declare a type *Time*, representing time values using the set of natural numbers. The SCJ API represents time as two numbers representing milliseconds and nanoseconds, but it is easier for the purposes of specification to ignore that detail, since a pair of numbers is only used in the SCJ API because Java has no type large enough to contain the information from both. Instead, we take *Time* values to represent the total number of nanoseconds.

$$Time == \mathbb{N}$$

The clock must have a *precision* value representing the number of nanoseconds between occurrences of the hardware clock interrupt. The *precision* cannot be zero.

$$\begin{array}{|l}
precision : Time \\
\hline
precision > 0
\end{array}$$

The SCJVM real-time clock relies on the existence of a hardware real-time clock that must be capable of giving the current time in nanoseconds. We declare the channel $HWtime$ for receiving the current value of the real-time clock from hardware.

$\mathbf{channel}\ HWtime : Time$

We also declare channels for each of the services of the real-time clock described in Section 3.3.

$\mathbf{channel}\ RTCgetTime, RTCgetPrecision : Time$
$\mathbf{channel}\ RTCsetAlarm : Time$
$\mathbf{channel}\ RTCclearAlarm$

The SCJVM also uses the hardware interrupt channel, $HWinterrupt$, and has a channel to pass the clock interrupt on to the scheduler when appropriate, $RTCclockInterrupt$, which was declared earlier. There is also a type, $RTCReport$, and channel, $RTCreport$, for reporting erroneous inputs to operations, as for the memory manager and scheduler.

Having defined the channels and types, the process definition can be presented.

$\mathbf{process}\ RealtimeClock \mathrel{\widehat=} \mathbf{begin}$

The real-time clock's state, $RTCState$, stores the current time value, $currentTime$, of the clock (accurate to within the clock's precision). The $RTCState$ also contains a component to represent the time $currentAlarm$ of the alarm set (if any) as well as a boolean component, $alarmSet$, indicating whether or not there is an alarm set. If an alarm is set, then it must be in the future.

$$\begin{array}{|l}
\hline
\textit{RTCState} \\
currentTime : Time \\
currentAlarm : Time \\
alarmSet : \mathbb{B} \\
\hline
alarmSet = True \Rightarrow currentAlarm \geq currentTime \\
\hline
\end{array}$$

This $RTCState$ schema is the state of the *Circus* process modelling the real-time clock.

$\mathbf{state}\ RTCState$

The clock's state is initialised with a time value, $initTime?$, to which the $currentTime$ is set. Initially no alarm is set, so $alarmSet$ is $False$ and $currentAlarm$ is allowed to take any value since it is unused.

$$\begin{array}{|l}
\hline
\textit{RTCInit} \\
RTCState' \\
initTime? : Time \\
\hline
currentTime' = initTime? \\
alarmSet' = False \\
\hline
\end{array}$$

The *initTime*? value is obtained from the hardware real-time clock via the *HWtime* channel.

$$Init \mathrel{\widehat{=}} HWtime?initTime \longrightarrow \big(RTCInit\big)$$

The operation of getting the clock's time value simply outputs *currentTime* on the *RTCgetTime* channel and then outputs a report of *RTCokay*.

$$GetTime \mathrel{\widehat{=}} RTCgetTime!currentTime \longrightarrow RTCreport!RTCokay \longrightarrow \mathbf{Skip}$$

The operation to get the clock's precision is similar. It outputs *precision* on the *RTCgetPrecision* channel.

The operation of setting a new alarm is described by *RTCSetAlarm*, which takes the time of the alarm, *alarmTime*?, as input. Since this is the only operation that has an error case, we do not have a separate lifting to robust operations, so we also provide a *report*! output. The *alarmTime*? must be greater than or equal to the *currentTime*, since an alarm cannot be set at a time in the past. The *currentAlarm* is set to the input *alarmTime* and *alarmSet* to **True**, since an alarm has been set. This operation does not affect the *currentTime*. We output the *report*! value *RTCokay* because this specifies the successful case of the operation.

$$\begin{array}{|l}
\multicolumn{1}{l}{RTCSetAlarm} \\
\hline
\Delta RTCState \\
alarmTime? : Time \\
report! : RTCReport \\
\hline
alarmTime? \geq currentTime \\
currentAlarm' = alarmTime? \\
alarmSet' = \mathbf{True} \\
currentTime' = currentTime \\
report! = RTCokay \\
\hline
\end{array}$$

The error case for this operation occurs if the given time is in the past. It is described by the schema *TimeInPast*, which has the same components as *RTCSetAlarm* but does not change the state. This case applies if *alarmTime*? is less than *currentTime* and results in a *report*! of *RTCtimeInPast*.

$$\begin{array}{|l}
\multicolumn{1}{l}{TimeInPast} \\
\hline
\Xi RTCState \\
alarmTime? : Time \\
report! : RTCReport \\
\hline
alarmTime? < currentTime \\
report! = RTCtimeInPast \\
\hline
\end{array}$$

The action that specifies the complete behaviour of the operation receives the alarm time via the *RTCsetAlarm* channel, and behaves as the disjunction of *RTCSetAlarm* and *TimeInPast*. The *report* output is communicated via the *RTCreport* channel.

$$\begin{array}{l}
SetAlarm \mathrel{\widehat{=}} \mathbf{var}\ report : RTCReport \bullet \\
\quad RTCsetAlarm?alarmTime \longrightarrow \big(RTCSetAlarm \lor TimeInPast\big); \\
\quad RTCreport!report \longrightarrow \mathbf{Skip}
\end{array}$$

The operation of clearing the alarm is defined using a *Circus* assignment action to set the *alarmSet* flag to *False* in response to a signal on the *RTCclearAlarm* channel. The *currentAlarm* value can be left at its previous value, since it is not used when there is no alarm set. This operation ends with a report of *RTCokay*.

$$ClearAlarm \mathrel{\widehat=} RTCclearAlarm \longrightarrow alarmSet := False \mathrel{;} RTCreport!RTCokay \longrightarrow \mathbf{Skip}$$

This concludes the definition of the public services of the real-time clock. The clock must also respond to triggering of alarms and hardware clock interrupts When an alarm triggers, the clock interrupt is sent to the scheduler via the *RTCclockInterrupt* channel, as described by the *TriggerAlarm* action. The current alarm is then cleared by setting *alarmSet* to *False*.

$$TriggerAlarm \mathrel{\widehat=} RTCclockInterrupt \longrightarrow alarmSet := False$$

Clock tick interrupts, which come periodically from the hardware with a period equal to *precision* are handled as described by *Tick*. The interrupts come via the *HWinterrupt* channel and are required to have the identifier of the clock interrupt (the non-clock interrupts are handled by the scheduler). An if statement is used to check if the *currentTime* with the *precision* value added to it is greater than *currentAlarm*. If it is, then the alarm triggers as described in *TriggerAlarm*. The *currentTime* value is then incremented by the clock's precision. Resolving alarms before updating *currentTime* is required to ensure the state invariant is maintained.

$$\begin{array}{l}
Tick \mathrel{\widehat=} HWinterrupt?interrupt : (interrupt = clockInterrupt) \longrightarrow \\
\quad \mathbf{if}\ currentTime + precision \geq currentAlarm \longrightarrow TriggerAlarm \\
\quad [\!]\ currentTime + precision < currentAlarm \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi} \mathrel{;} currentTime := currentTime + precision
\end{array}$$

Any of the actions available to the user may be chosen, and the process loops to allow another action to be taken. The process may handle an incoming clock tick instead of a user action.

$$Loop \mathrel{\widehat=} SetAlarm \mathbin{\Box} ClearAlarm \mathbin{\Box} GetTime \mathbin{\Box} GetPrecision \mathbin{\Box} Tick \mathrel{;} Loop$$

The main action of the process begins by performing the initialisation and then enters the loop.

$$\bullet\ Init \mathrel{;} Loop$$

$$\mathbf{end}$$

We have now specified the real-time clock that tracks the current time and any alarm that may be set. Operations are provided to set and clear the alarm. The state of the clock is updated when a clock interrupt signal is received and the clock is checked against the alarm, forwarding the interrupt signal to the scheduler if the alarm time has passed.

3.4.4 Complete VM Services Model

Having defined the three processes that model the three components of the VM services, we now compose them in parallel to form the complete model of the VM services.

Certain channels are used to communicate between the different components. The channel set *RTCSInterface* contains the channel used to pass the clock interrupt from the real-time clock to the scheduler.

$$\textbf{channelset}\ RTCSInterface == \{\!|\ RTCclockInterrupt\ |\!\}$$

We define the *VMServices* process by composing each of the components of the SCJVM services in parallel, with the *Scheduler* and *RealtimeClock* synchronising on *RTCSInterface*, which is then hidden since it is an internal channel.

$$\begin{array}{l}\textbf{process}\ VMServices \mathrel{\widehat=} ((MemoryManager \interleave Scheduler)\\ \qquad [\![RTCSInterface\ ]\!]\ RealtimeClock)\\ \qquad\quad \setminus RTCSInterface\end{array}$$

So the *VMServices* process represents a complete model of the SCJVM services.

3.5 Final Considerations

In this chapter, we have presented the services that must be provided by an SCJVM in order to support the core execution environment and the SCJ API. We have divided these services into three areas, the memory manager, the scheduler, and the real-time clock, and detailed the services provided in each area. We have also presented our model of the SCJVM services in the *Circus* specification language, of which a full version can be found in Appendix A of the extended version of this thesis [13].

Our model is composed of a *Circus* process for each of the three classes of services we have identified. The memory manager process largely consists of Z data operations on the state of the memory, which are then lifted to *Circus* actions that can be accessed via channels. The scheduler also consists of a large Z model, but requires more reliance on *Circus* to specify interaction with interrupts. The real-time clock model is mainly made up of *Circus* actions with few Z schemas, though it is also a smaller component than the other two due to the small number of services it provides.

Note that our model is written as an abstract specification, so it records invariants and allows for nondeterminism that need not be present in an implementation, but are useful for reasoning about the model. As an example of this, we note that the memory manager model consists of 583 lines of code (as measured in *Circus* LaTeX syntax, ignoring blank lines and comments), whereas icecap's `vm.Memory` class[1], which contains the code for backing store operations, consists of just 238 lines of code (ignoring blank lines and comments). The SCJ memory manager implementation on JOP described in [106] is even smaller, containing just 148 lines of code (ignoring blank lines and comments)[2].

The reason these implementations are smaller than our model is that we have invariants specifying the relationship of the different components of the memory manager, particularly the invariants of *GlobalMemoryManager* and *GlobalStoresManager*, which specify the relationships between backing stores. In addition, `vm.Memory` from icecap represents a single backing store, corresponding to the *BackingStore* type in our model. The *GlobalMemoryManager* is not explicitly

[1]The `vm.Memory` class can be found at `https://github.com/scj-devel/hvm-scj/blob/master/icecapSDK/src/vm/Memory.java`

[2]The class containing the JOP SCJ memory manager implementation can be found at `https://github.com/jop-devel/jop/blob/master/java/target/src/common/com/jopdesign/sys/Memory.java`

implemented in icecap; instead the *stores* map (which is a component of *GlobalMemoryManager*) corresponds to the set of pointers to all the valid instances of `vm.Memory`, with the instances at those pointers forming the range of the map. The mapping specified by the *stores* map thus captures the implicit mapping between a pointer and the object to which it points. The *childRelation* component of the *GlobalMemoryManager* exists only to specify the structure of *stores*, so it is not present in icecap either, while the *rootBackingStore* identifier corresponds to the `vm.Memory` pointer stored in the `ImmortalMemory`. While its explicit inclusion in icecap is not necessary, the inclusion of the *GlobalMemoryManager* in our model allows us to specify the global structure of backing stores and ensure that each of the operations preserve that structure.

Our memory manager model also includes nondeterminism. For example, we avoid specifying at which end of a backing store nested backing stores should be allocated. This results in a more verbose specification in terms of sets of memory, but avoids unduly constraining an implementation. Similar considerations apply to the other parts of our model.

Overall, the division of the SCJVM services into the three areas we have chosen appears to give a good separation between the components with little coupling. This is shown in Figure 3.3, where it can be seen that only one channel, *RTCclockInterrupt*, is required for communication between the processes in the model. The use of *Circus* has allowed us to specify the few necessary points of communication between these processes, and also their relation to hardware interrupts and the core execution environment.

The fact that the requirements of scheduler and memory manager model are largely expressed in Z allows them to be checked using Z proof tools. Indeed, we have already partially subjected the memory manager model to proof using Z/Eves. The proofs we have performed are consistency proofs and proofs that functions are not applied outside their domain. We have performed these proofs for the first part of the memory manager model, covering memory blocks, and also partially for backing stores. The theorems we have proved about the memory manager, along with their proofs and some additional lemmas about mathematical toolkit objects that we have proved in the course of our work, can be found in Appendix E of the extended version of this thesis [13].

Our formal model of the SCJVM services supports our specification of the core execution environment described in the next chapter, which forms the basis for our compilation strategy, described in Chapter 5. The model of the SCJVM services can be used to create an SCJVM services implementation by refinement from the model, which can be translated to executable code. This implementation can support the output of the compilation strategy. Properties proved for the specification of the SCJVM services are preserved by the refinement and so it can be known that those properties also hold for the implementation.

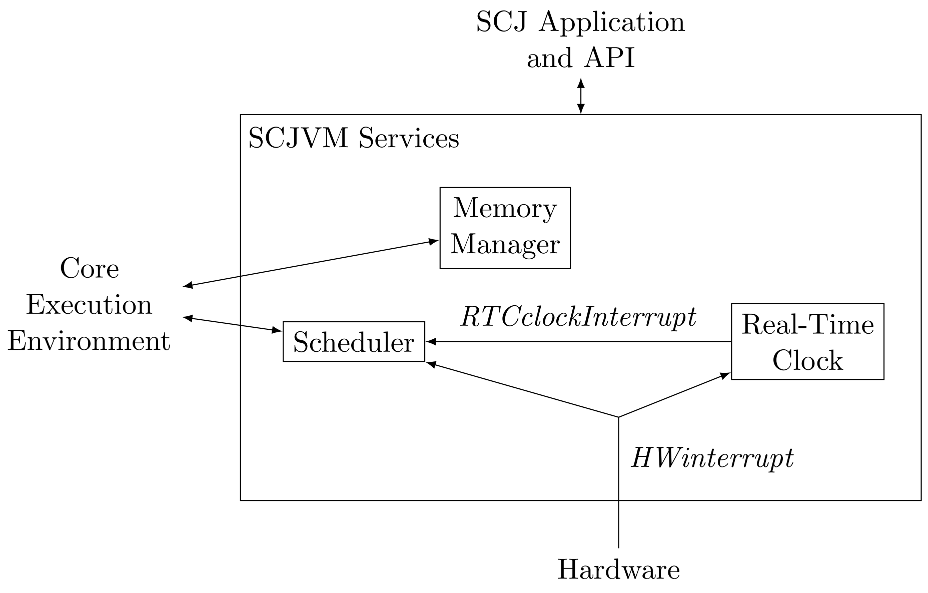


Figure 3.3: The structure of the SCJVM services model, showing the channels used for communication between the processes in the model

Chapter 4

The Core Execution Environment

This chapter describes the core execution environment (CEE) of an SCJVM, which handles execution of an SCJ program. In addition, the CEE of an SCJVM manages the flow of execution dictated by the SCJ programming model, including, for example, `Safelet` setup and mission execution.

This is the part of our SCJVM model that is handled by our compilation strategy. So, it may take the form of a bytecode interpreter, which is the starting point for the compilation, or C code, which is the output of the compilation. We describe both of these in this chapter (Sections 4.2, 4.3 and 4.4) while the compilation strategy for transforming between them is described in the next chapter. We begin with an overview of the CEE's structure in the next section. After the presentation of our model, we discuss how it is validated in Section 4.5 and then conclude with some final considerations in Section 4.6.

4.1 Overview

The CEE has three components, two of which depend on whether it is interpreting bytecodes or executing C code. For the CEEs that use a bytecode interpreter, the components are listed below and shown in Figure 4.1:

- the object manager, which manages information about objects created during execution of the bytecode;
- the interpreter itself, which handles execution of bytecode instructions; and
- the launcher, which coordinates the startup of the SCJVM, the execution of missions, and the execution of methods in the interpreter.

The components after compilation to C are similar, but the object manager is replaced with a struct manager, which manages C struct types representing objects, and the interpreter is replaced with the C program itself. The launcher remains unchanged throughout the compilation. It is assumed that it is already in the form of native code that can be called from the C code.

The CEE is combined with the SCJVM services to form the complete SCJVM; this is indicated in Figure 4.1, which shows the same structure described in Figure 3.1 in the previous chapter,

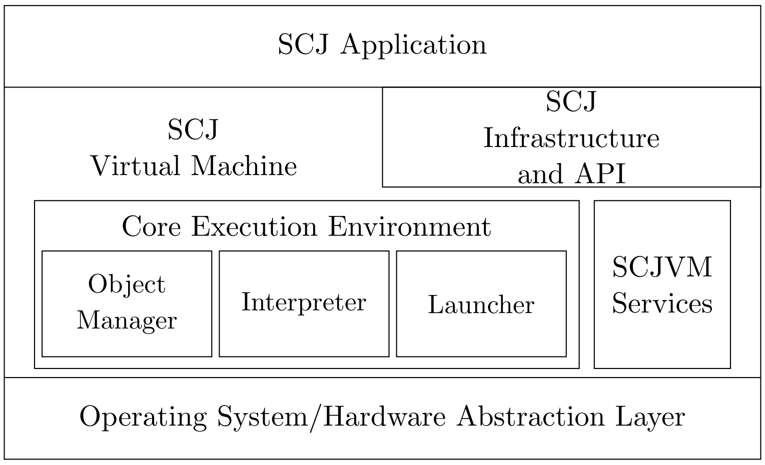

SCJ Application
SCJ Virtual Machine
SCJ Infrastructure and API
Core Execution Environment
Object Manager
Interpreter
Launcher
SCJVM Services
Operating System/Hardware Abstraction Layer

Figure 4.1: Structure of an SCJVM, showing the components of the CEE, and its relation to the SCJ infrastructure and the operating system/hardware abstraction layer

but has a focus on the CEE components. The SCJVM services are unaffected by the compilation strategy and can be implemented as a separate library.

Each of the components of the CEE is represented by a single *Circus* process in our model. These processes interact as shown in Figure 4.2. The overall pattern of the interaction is unaffected by the compilation, that is, the model of the compiled code has the same overall flow of communication, although the components have different names.

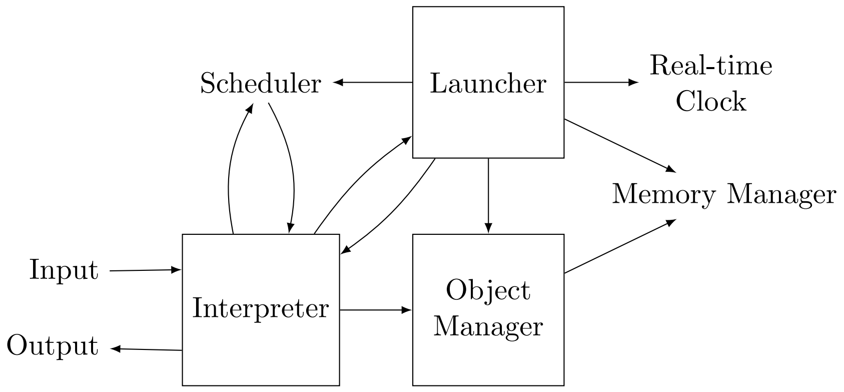


Figure 4.2: The CEE model processes and their communication with each other and the SCJVM services

The launcher manages the startup procedure for the SCJVM and the execution of missions. This involves communication with the interpreter (or C program) to execute initialisation methods. Allocation of backing stores for the schedulable objects and entering the corresponding memory areas involves communication with both the object (or struct) manager in the CEE and the memory manager of the SCJVM services. The launcher must also communicate with the scheduler to indicate when threads should be started or suspended during mission execution, and with the real-time clock to set alarms coordinating event-handler execution.

The interpreter must accept the requests to execute methods on the main thread from the launcher, and it must also respond to requests from the scheduler to start the other threads. When a thread has finished execution, the interpreter signals to the scheduler that the thread has finished so that it is no longer scheduled. The interpreter must also communicate with the launcher to handle calls to methods that are provided by the SCJ infrastructure, such as the methods to enter memory areas. Handling of memory allocation during method execution is performed via communication with the object manager, which then communicates with the SCJVM memory manager. Additionally, the interpreter communicates inputs and outputs to some console input/output device, which is the only such device required by the SCJ specification. Supporting a full range of hardware connections is beyond the scope of this work.

The interactions just described are modelled by channel communications. Those with the SCJVM services memory manager and scheduler use the channels already described in Sections 3.4.1 and 3.4.2. The channels used for communication between the CEE processes are summarised in Table 4.1, with the full channel declarations shown in Appendix B of the extended version of this thesis [13]. In addition to presenting the name and type for each channel, in the first two columns of the table, we also indicate which components of the CEE make use of the channel, in the third and fourth columns of the table. The channels *output* and *input* are used for communication with the console device mentioned earlier. As we do not model the console device itself, these are left as externally visible channels when the component processes are composed into the complete SCJVM model. Some channels are marked with various symbols (*, †, § and +) so that we can refer to them later in the text.

The types of values communicated by those channels are also used by the CEE processes. These include the type of object identifiers, *ObjectID*, the type of thread identifiers, *ThreadID*, the type of backing store identifiers, *BackingStoreID*, and the type of virtual machine data words, *Word*. We also use the *ClassID* and *MethodID* types, which are the types of class and method identifiers that are declared in the scheduler model to permit the declaration of the *CEEstartThread* channel. Additionally, we declare a field identifier type, *FieldID*.

$$[\mathit{FieldID}]$$

The class, method and field identifiers may be the full names used in Java class files or some shorter representation, such as unique identification numbers. In any case, type information needs to be taken into account so that methods and fields with the same name, but different type signatures, have different identifiers. This is because the identifiers in Java class files include the type information and the correct operation of method overloading relies on it.

We also declare a set, *initialisationMethodIDs*, of method identifiers, which contains those method identifiers that refer to instance initialisation methods (i.e. constructors, recognisable as having the name `<init>` in Java class files). This information is used by the `invokespecial` instruction, which determines the target of non-initialisation superclass methods in a different way to that of other methods.

$$\mid \; \mathit{initialisationMethodIDs} : \mathbb{P}\, \mathit{MethodID}$$

Most of the channels are part of pairs, with one channel to communicate a signal to begin an operation and supply any inputs, and a return channel to communicate back when the operation has finished and supply any outputs. The return channel is named by appending *Ret* to the name of the channel used to initiate the operation. For brevity, we omit return channels with no parameters from Table 4.1 and mark the channels having such a return channel with a *.

The *executeMethod* channel is used to signal to the interpreter that it should begin execution of a method on a given thread. The interpreter signals on the *executeMethodRet* channel when it has finished execution of the method. For methods executed on a thread other than the main

	Name	Parameter Type	Communication from	to
	executeMethod	$ThreadID \times ClassID \times MethodID \times \text{seq}\ Word$	L	I
	executeMethodRet	$ThreadID \times Word$	I	L
	continue	$ThreadID$	L	I
	endThread	$ThreadID$	L	I
	initMainThread	$StackID$	L	I
	runThread	$ThreadID \times ObjectID \times MethodID$	I	L
*†	*register*	$ThreadID \times ObjectID$	I	L
*†	*releaseAperiodic*	$ObjectID$	I	L
*†	*enterPrivateMemory*	$ThreadID \times \mathbb{N}$	I	L
*†	*executeInAreaOf*	$ThreadID \times ObjectID$	I	L
*†	*executeInOuterArea*	$ThreadID$	I	L
*†	*exitMemory*	$ThreadID$	I	L
*§	*takeLock*	$ObjectID$	I	L
*§	*releaseLock*	$ObjectID$	I	L
*§	*setPriorityCeiling*	$ObjectID \times Priority$	I	L
*+	*initAPEH*	$ThreadID \times ObjectID \times Priority \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$	I	L
*+	*initPEH*	$ThreadID \times ObjectID \times Priority \times Time \times Time \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$	I	L
*+	*initOSEHAbs*	$ThreadID \times ObjectID \times Priority \times Time \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$	I	L
*+	*initOSEHRel*	$ThreadID \times ObjectID \times Priority \times Time \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$	I	L
	output	$Word$	I	<ext.>
	input	$Word$	<ext.>	I
	enterBackingStore	$ThreadID \times BackingStoreID$	L	OM
	exitBackingStore	$ThreadID$	L	OM
	exitBackingStoreRet	$BackingStoreID \times \mathbb{B}$	L	OM
	getCurrentAC	$ThreadID$	L	OM
	getCurrentACRet	$BackingStoreID$	OM	L
	newObject	$ThreadID \times ClassID$	I/L	OM
	newObjectRet	$ObjectID$	OM	I/L
	getClassIDOf	$ObjectID \times ClassID$	I/L	OM
	getField	$ObjectID \times ClassID \times FieldID$	I	OM
	getFieldRet	$Word$	OM	I
	putField	$ObjectID \times ClassID \times FieldID \times Word$	I	OM
	getStatic	$ClassID \times FieldID$	I	OM
	getStaticRet	$Word$	OM	I
	putStatic	$ClassID \times FieldID \times Word$	I	OM
	addThreadMemory	$ThreadID \times BackingStoreID$	I	OM
	removeThreadMemory	$ThreadID$	I	OM

Table 4.1: The channels used for communication between CEE processes before compilation. In the final two columns, L refers to the launcher, I refers to the interpreter, OM refers to the object manager, I/L indicates a channel shared by the interpreter and launcher in interleaving, and <ext.> indicates an external channel.

thread, we use two further channels, *continue* and *endThread*, to indicate whether a thread should continue to accept method executions or finish accepting method executions respectively.

Before the interpreter can execute methods on the main thread, its stack space must allocated by the launcher and communicated to the interpreter. This is handled by the *initMainThread* channel, which carries the *StackID* for the stack space allocated for the main thread. The stack for non-main threads is supplied from the scheduler using the *CEEstartThread* channel.

When a thread other than the main thread starts running, the *Interpreter* is signalled by the scheduler and begins executing the thread. However, the entry points of most threads are handled in the *Launcher*, since event-handler threads, for example, form part of the SCJ infrastructure. A channel, *runThread*, is thus used to pass execution from the *Interpreter* to the *Launcher* after initial setup of the thread has been completed. This carries the *ThreadID* of the thread, along with the identifier of the object representing the thread (either an event handler or an interrupt handler) and the identifier of the method to be executed (either `run()` or `handle()`). The *Launcher* then handles the execution of the thread, calling back to the *Interpreter* using the *executeMethod* channel when necessary to execute programmer-supplied code in the handler.

As mentioned above, while executing a method, the interpreter may signal back to the launcher for handling of special methods. The channels used for this are the ones marked with a +, a †, or a § in Table 4.1. The channels marked with a † are used to implement calls to infrastructure methods that are part of the SCJ API. The inputs and outputs of these methods (and hence the types of the channels associated with them) are taken from the SCJ specification.

The methods for entering memory areas are handled in a slightly different way. In their case, the identifier of the `Runnable` object passed to a method is not communicated on the channels associated with their calls. In response to a signal on those channels, the launcher instead enters the appropriate memory area, and the `Runnable` object's `run()` method is called by executing a bytecode instruction in the interpreter. Another special method is used to exit the memory area after the `run()` method returns. It is called using the *exitMemory* channel. This approach simplifies the interaction between the launcher and interpreter models, since it avoids call backs.

The channels marked with a + are used to implement special methods for the intialisers of the event handler objects. They take as parameters the information passed to the constructors in the SCJ API. The event handler constructors accept parameters in the form of objects of classes that group together information such as the storage sizes and scheduling times. For simplicity, the channels used for the event handler intialisers accept the information directly, without the added structure of these objects. We also provide separate channels for initialisation of one-shot event handlers, depending on whether they are initialised with an absolute or relative start time.

We only consider constructors for the `AperiodicEventHandler`, `PeriodicEventHandler`, and `OneShotEventHandler` classes. We also do not consider deadline miss handlers. This is because none of these features affect the compilation strategy, so they are not needed to evaluate it. A complete formal account of the execution model for the event handlers is available elsewhere [26, 68, 119].

The channels marked with a § expose SCJVM scheduler operations to the code executed in the interpreter. Their types follow those of the scheduler's channels.

The *output* and *input* channels are used to communicate *Word* values to and from a console device. The rest of the channels are used by the launcher and the interpreter to communicate with the object manager. The *enterBackingStore* channel is used by the launcher to signal to the object manager when a memory area is entered so that it can record that the corresponding backing store has been entered. This carries the *ThreadID* of the thread to be entered, since the backing stores entered are recorded separately for each thread, and the *BackingStoreID* of the backing store to be entered. There is no corresponding return channel, since it is not necessary for the launcher to wait while the object manager records the entry to a backing store. Similarly, the *exitBackingStore* channel is used to signal an exit from the backing store that is the current allocation context of the given thread. This does have a return channel, since the launcher must be informed if the backing store was cleared due to no longer being in use by any thread. Additionally, the *getCurrentAC* channel (and its return channel) is used to obtain the *BackingStoreID* of the backing store used as the current allocation context for a given thread from the object manager.

The remaining channels used by the launcher to communicate with the object manager are used by both the launcher and the interpreter. These are the *newObject* channel, which is used to allocate space for new objects in the current allocation context, and the *getClassIDOf* channel, which is used to obtain the *ClassID* for the class of the object associated with a given *ObjectID*. The *newObject* channel carries the *ThreadID* of the current thread, since there is a separate allocation context for each thread, and the *ClassID* of the class of the object to be allocated. The object manager returns the *ObjectID* of the newly allocated object via the corresponding return channel. The *getClassIDOf* channel carries both the input and output to the operation on the same channel, since it is a simple data accessing operation that can be dealt with in a single communication.

The other channels used by the interpreter are the channels for accessing fields of objects and classes. The *getField* channel is used for obtaining the value stored in a given field of a given object. It carries the *ObjectID* of the object whose field is to be accessed, the *ClassID* of the object's class and the *FieldID* of the field to be accessed. The object manager then returns the *Word* value stored in the field. For putting a value into an object's field, the *putField* channel is used, which carries the *Word* value to store in the field in addition to the *ObjectID*, *ClassID* and *FieldID* that identify the object and field to update. As this just updates the field and does not return any information, there is no need for a return channel. Channels for accessing static fields, *getStatic* and *putStatic*, are also provided. These operate similarly to the channels for object fields but use *ClassID* values rather than *ObjectID* values, since static fields are attached to classes rather than objects.

The final channels used by the interpreter are the *addThreadMemory* and *removeThreadMemory* channels. The *addThreadMemory* channel is used to inform the object manager of a thread's initial allocation context when the thread starts. It carries the *ThreadID* of the thread and the *BackingStoreID* of the backing store that serves as the thread's initial allocation context. When a thread has finished execution, it informs the object manager via the *removeThreadMemory* channel, which carries the *ThreadID* of the thread.

Next, in Section 4.2, we describe our model of the launcher. We then detail the bytecode interpreter model in Section 4.3, and the C code model in Section 4.4.

4.2 Launcher

As mentioned in the previous section, the launcher is the component of the CEE that manages the SCJVM startup and coordinates mission execution. It is described by the *Launcher* process.

The launcher remains unaffected throughout the compilation strategy, because it is agnostic to the class and bytecode information. However, the launcher must know where to begin execution, so it takes a parameter, *safeletClass*, which is the *ClassID* of the `Safelet` class. This can be seen in the the *Launcher* process definition, the beginning of which is shown below.

Class initialisers must be executed as part of the SCJVM startup procedure. The order in which they are executed is determined by the dependencies between class initialisers and classes, and is also passed to the *Launcher* process as a second parameter, *initOrder*, which is a sequence of *ClassID*s.

$$\mathbf{process}\, \mathit{Launcher} \mathrel{\widehat=} \mathit{safeletClass} : \mathit{ClassID};\ \mathit{initOrder} : \mathrm{seq}\, \mathit{ClassID} \bullet \mathbf{begin}$$

In what follows, we describe the definition of *Launcher*, focusing on the aspects relevant for the compilation. The complete definition can be found in Appendix B of the extended version of this thesis [13].

The state of the *Launcher* is divided into four parts. The first part contains the identifiers of the objects that form the SCJ mission model, so that the *Launcher* can call methods of those objects during the SCJVM startup. The second part contains information on the memory-area objects, including the relationship between the memory areas and the backing stores they represent, so that methods for entering and exiting memory areas can be handled. The third part of the state contains information about the schedulable objects of SCJ and the threads used by the CEE so that the schedulable objects can be managed by the *Launcher*. The final part contains information on alarms for coordinating releases of schedulable objects, so that they can be set in the real-time clock in the correct order.

We use separate Z schemas to specify each part of the state. The first part is described by the *MissionManager* schema, shown below. It contains the identifiers of three objects:

- *safelet*, the instance of the class implementing the `Safelet` interface for the program;
- *missionSequencer*, the mission sequencer returned by the `getSequencer()` method of the safelet; and
- *currentMission*, the mission that is currently executing.

Methods of these objects are called at various points throughout SCJVM startup and mission execution.

```
  MissionManager ____________________________________________
 | safelet, missionSequencer, currentMission : ObjectID
 |___________________________________________________________
```

The second part of the *Launcher*'s state is described by the *MemoryAreaManager* schema below. It contains the memory-area object identifiers for the immortal memory, *immortalMemory*, and mission memory, *missionMemory*. There is a map, *backingStores*, that relates these identifiers and the identifiers of the other memory-area objects, to the identifiers of the backing stores they represent. We also record the backing store identifiers of the per-release memories for each thread in the *perReleaseMemories* map. Finally, to make sure that nested private memories can be reused, there is a map from backing store identifiers to the identifiers of private backing stores they contain, *privateMemoryMap*.

$$
\begin{array}{|l}
\textit{MemoryAreaManager} \\
\hline
\mathit{immortalMemory}, \mathit{missionMemory} : \mathit{ObjectID} \\
\mathit{backingStores} : \mathit{ObjectID} \pinj \mathit{BackingStoreID} \\
\mathit{perReleaseMemories} : \mathit{ThreadID} \pinj \mathit{BackingStoreID} \\
\mathit{privateMemoryMap} : \mathit{BackingStoreID} \pinj \mathit{BackingStoreID} \\
\hline
\mathit{immortalMemory} \neq \mathit{null} \Rightarrow \mathit{immortalMemory} \in \operatorname{dom} \mathit{backingStores} \\
\mathit{missionMemory} \neq \mathit{null} \Rightarrow \mathit{missionMemory} \in \operatorname{dom} \mathit{backingStores} \\
\operatorname{ran} \mathit{perReleaseMemories} \subseteq \operatorname{ran} \mathit{backingStores} \\
\mathit{backingStores}^{\sim} \llparenthesis \operatorname{ran} \mathit{perReleaseMemories} \rrparenthesis \\
\qquad \cap \{\mathit{immortalMemory}, \mathit{missionMemory}\} = \varnothing \\
\mathit{immortalMemory} \neq \mathit{missionMemory} \\
\mathit{id}\ \mathit{BackingStoreID} \cap \mathit{privateMemoryMap}^{+} = \varnothing \\
(\mathit{backingStores} \llparenthesis \{\mathit{immortalMemory}, \mathit{missionMemory}\} \rrparenthesis \\
\qquad \cup \operatorname{ran} \mathit{perReleaseMemories}) \cap \operatorname{ran} \mathit{privateMemoryMap} = \{\} \\
\hline
\end{array}
$$

Each of the maps in *MemoryAreaManager* is injective, since each memory-area object has a distinct backing store and memory areas cannot share a nested private memory area. The invariant of *MemoryAreaManager* requires that the *immortalMemory* and *missionMemory*, when they are initialised, as well as each of the *perReleaseMemories*, must have a corresponding backing store in *backingStores*. It also requires that *immortalMemory* and *missionMemory* are not in *perReleaseMemories* and are not equal to one another. Together with the injectivity requirement on *perReleaseMemories*, this ensures each memory area is distinct. Finally, the invariant requires that private memories cannot be nested inside themselves, and that *immortalMemory*, *missionMemory* and all the *perReleaseMemories* are not nested private memories. The first of these is specified by taking the transitive closure of *privateMemoryMap* to obtain the mapping from backing stores to backing stores that are (directly or indirectly) nested within them. This is then required to be disjoint from the identity map from a backing store to itself. The second of these properties is specified in a similar way, stating that the backing stores corresponding to *immortalMemory*, *missionMemory* and the *perReleaseMemories* must be disjoint from those in the range of *privateMemoryMap*.

The third part of the state is specified in the *SchedulableManager* schema below. It contains maps that relate the identifiers of schedulable objects to information about them. The *schedulableThreads* map records the thread identifier for each schedulable object. This map must be injective, since every schedulable object has a separate thread. The *schedulableTypes* map has information about which type of event handler each schedulable object belongs to. This is represented by a type *HandlerType* that has constructors representing aperiodic, periodic, and one-shot event handlers. The *schedulableSizes* map records the backing store, allocation area, and stack sizes required for each schedulable object. In addition to these maps, *SchedulableManager* contains a set *registeredObjects*, containing the schedulable objects that have been registered as being part of the current mission.

$$
\begin{array}{|l}
\textit{SchedulableManager} \\
\hline
\mathit{schedulableThreads} : \mathit{ObjectID} \pinj \mathit{ThreadID} \\
\mathit{schedulableTypes} : \mathit{ObjectID} \pfun \mathit{HandlerType} \\
\mathit{schedulableSizes} : \mathit{ObjectID} \pfun (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
\mathit{registeredObjects} : \mathbb{F}\ \mathit{ObjectID} \\
\hline
\operatorname{dom} \mathit{schedulableThreads} = \operatorname{dom} \mathit{schedulableTypes} = \operatorname{dom} \mathit{schedulableSizes} \\
\mathit{registeredObjects} \subseteq \operatorname{dom} \mathit{schedulableThreads} \\
\hline
\end{array}
$$

The invariant of *SchedulableManager* ensures the maps all have the same domain and that *registeredObjects* is a subset of the map domains.

The final part of the state is described by the *AlarmManager* schema below. It contains a map, *alarms*, that records the time of the next alarm to be set for each thread. It also stores the identifier, *clockHandler*, of the clock interrupt handler object, since that is used in management of alarms.

$$\begin{array}{|l}
\textit{AlarmManager} \\ \hline
\textit{alarms} : \textit{ThreadID} \pfun \textit{Time} \\
\textit{clockHandler} : \textit{ObjectID} \\ \hline
\end{array}$$

The state of the process is then the conjunction of the four schemas above.

$$\begin{array}{l}
\textbf{state}\ \textit{LauncherState} == \\
\quad \textit{MissionManager} \land \textit{MemoryAreaManager} \land \textit{SchedulableManager} \land \textit{AlarmManager}
\end{array}$$

The *Launcher* state is initialised as described in *LauncherInit*, shown below. The object identifiers are initialised to the *null* identifier. They are later updated as the corresponding objects are created during SCJVM execution. Similarly, each of the maps and sets is initialised to the empty set.

$$\begin{array}{|l}
\textit{LauncherInit} \\ \hline
\textit{LauncherState}' \\ \hline
\{\textit{safelet}', \textit{missionSequencer}', \textit{currentMission}', \textit{immortalMemory}', \\
\quad \textit{missionMemory}', \textit{clockHandler}'\} \\
\quad \subseteq \{\textit{null}\} \\
\textit{backingStores}' = \varnothing \\
\textit{perReleaseMemories}' = \varnothing \\
\textit{privateMemoryMap}' = \varnothing \\
\textit{registeredObjects}' = \varnothing \\
\textit{schedulableThreads}' = \varnothing \\
\textit{schedulableTypes}' = \varnothing \\
\textit{schedulableSizes}' = \varnothing \\
\textit{alarms}' = \varnothing \\ \hline
\end{array}$$

The main action of the *Launcher* proceeds as shown below. The state is first initialised as described by *LauncherInit* and then the actions *Startup* and *RunNextMission* follow in sequence. *Startup* defines the SCJVM startup procedure that must be performed once at the start of SCJVM execution, whereas *RunNextMission* defines the procedure performed for each mission run. We do not handle mission termination. This is because the SCJ mission termination procedure has almost no effect on our compilation strategy; a single mission is sufficient to evaluate the compilation strategy. A formal account of it is available elsewhere [26, 68, 119]. Thus, *RunNextMission* is only executed once.

$$\bullet\ \big(\textit{LauncherInit}\big)\ ;\ \textit{Startup}\ ;\ \textit{RunNextMission}$$

The definition of *Startup* is shown below. It performs a number of actions in sequence, following the startup procedure for an SCJVM:

- creating the main thread's stack and communicating it on the *initMainThread* channel, in *MakeMainStack*;
- executing the class initialisers in the order given in *initOrder*, in *RunClassInitialisers*;
- creating the immortal memory object that corresponds to the root backing store and storing it in *immortalMemory*, in *CreateImmortalMemory*;
- creating the `Safelet` object and storing it in *safelet*, in *CreateSafelet*;
- calling the safelet's `immortalMemorySize()` and `globalBackingStoreSize()` methods, and checking that the size of the root backing store matches the returned values, resizing it to match the result of the first method if possible, in *CheckImmortalMemory* and *CheckRemainingBackingStore*;
- calling the `initializeApplication()` method of the *safelet*, in *InitializeApplication*;
- creating the clock interrupt handler object, storing it in *clockHandler*, and registering it with the scheduler, in *MakeClockHandler*;
- calling the `getSequencer()` method of the *safelet* and storing the returned value in *missionSequencer*, in *GetSequencer*; and
- creating the *missionMemory* object and its backing store, in *CreateMissionMemory*.

$$\begin{aligned}&\mathit{Startup} \mathrel{\widehat=} \mathit{MakeMainStack} \mathbin{;} \mathit{RunClassInitialisers} \mathbin{;} \mathit{CreateImmortalMemory};\\&\quad \mathit{CreateSafelet} \mathbin{;} \mathit{CheckImmortalMemory} \mathbin{;} \mathit{CheckRemainingBackingStore};\\&\quad \mathit{InitializeApplication} \mathbin{;} \mathit{MakeClockHandler} \mathbin{;} \mathit{GetSequencer} \mathbin{;} \mathit{CreateMissionMemory}\end{aligned}$$

RunNextMission begins with calling the `getNextMission()` method of *missionSequencer*, in the action *GetNextMission*. The returned mission is stored in *currentMission*. Then, its `missionMemorySize()` method is executed, and the backing store of *missionMemory* is resized to match, in *ResizeMissionMemory*. Next, in *InitializeMission*, the mission's `initialize()` method is executed, during which the schedulable objects for the mission are registered. Afterwards, in *InitialiseAndStartThreads*, the registered schedulable objects have their stacks and backing stores created, after which the threads for all the schedulable objects are started. Finally, in *WaitForExecution*, the *main* thread suspends itself and the *Launcher* then waits, managing the threads of the program and handling special methods as necessary. Since termination is not handled, this phase of the program continues indefinitely.

$$\begin{aligned}&\mathit{RunNextMission} \mathrel{\widehat=} \mathit{GetNextMission} \mathbin{;} \mathit{ResizeMissionMemory} \mathbin{;} \mathit{InitializeMission};\\&\quad \mathit{InitialiseAndStartThreads} \mathbin{;} \mathit{WaitForExecution}\end{aligned}$$

During these actions, methods are executed using the *executeMethod* and *executeMethodRet* channels, discussed earlier. The identifiers of the methods, which may be standard methods from the SCJ API, or implementation-defined API methods required by the launcher, are represented by constants in the model. Although most of the methods used by the *Launcher* are executed simply by communicating on each of the channels mentioned above in turn, in *InitializeMission* the `initialize()` method of a mission requires handling of the `register()` method for each schedulable object. We must also provide handling for the special methods mentioned in the previous section. This is done in the *HandleSpecialMethodsMainLoop* action below, which offers handling of the special methods while waiting for return from the `initialize()` method on the *executeMethodRet* channel. A similar action, without the final choice accepting *executeMethodRet*, is used to handle special methods in *WaitForExecution*.

$$
\begin{array}{l}
\mathit{HandleSpecialMethodsMainLoop} \mathrel{\widehat=} \\
\quad \mathbf{val}\, \mathit{memoryEntries} : \mathit{ThreadID} \to \mathbb{N};\ \mathbf{res}\, \mathit{retVal} : \mathit{Word} \bullet \\
\quad (\Box\, t : \mathit{ThreadID} \bullet \\
\qquad \mathit{EnterMemory}(t); \\
\qquad \mathit{HandleSpecialMethodsMainLoop}(\\
\qquad\quad \mathit{memoryEntries} \oplus \{t \mapsto \mathit{memoryEntries}\, t + 1\}, \mathit{retVal})) \\
\quad \Box \\
\quad (\Box\, t : \mathit{ThreadID} \bullet \\
\qquad (\mathit{memoryEntries}\, t > 0) \mathbin{\&} \mathit{ExitMemory}(t); \\
\qquad \mathit{HandleSpecialMethodsMainLoop}(\\
\qquad\quad \mathit{memoryEntries} \oplus \{t \mapsto \mathit{memoryEntries}\, t - 1\}, \mathit{retVal})) \\
\quad \Box \\
\quad ((\mathit{Register} \Box \mathit{TakeLock} \Box \mathit{ReleaseLock} \Box \mathit{SetPriorityCeiling} \\
\qquad \Box\, \mathit{InitAperiodicEventHandler} \Box \mathit{InitPeriodicEventHandler} \\
\qquad \Box\, \mathit{InitOneShotEventHandlerRel} \Box \mathit{InitOneShotEventHandlerAbs}); \\
\qquad \mathit{HandleSpecialMethodsMainLoop}(\mathit{memoryEntries}, \mathit{retVal})) \\
\quad \Box \\
\quad ((\forall\, t : \mathit{ThreadID} \bullet \mathit{memoryEntries}\, t = 0) \mathbin{\&} \\
\qquad \mathit{executeMethodRet}?\mathit{thr} : (\mathit{thr} = \mathit{main})?r \longrightarrow \mathit{retVal} := r)
\end{array}
$$

HandleSpecialMethodsMainLoop takes a value parameter, *memoryEntries*, which is a map recording how many times a memory area has been entered for each thread. It also has a result parameter, *retVal*, which captures the return value from the execution of the method on the *main* thread. It offers a choice of handling a memory-area entry, handling the corresponding memory-area exit, handling a special method that does not enter memory areas, or accepting return from the execution of the method on the *main* thread (handled in the usual way using *executeMethodRet*, with the return value stored in *retVal*).

When *HandleSpecialMethodsMainLoop* handles memory-area entering methods, afterwards another method is executed in the interpreter, during which further special methods may be called. Each entry to a memory area must be matched by a corresponding exit from the memory area after this extra method execution returns. Thus, the entries to memory areas are tracked in the *memoryEntries* map.

The number stored in *memoryEntries* for a thread identifier t is incremented after handling a memory-area entry on that thread as described in *EnterMemory*(t). Similarly, it is decremented after handling exit from the memory area in *ExitMemory*(t), which is only offered if the value is already greater than zero. After handling memory-area entry or exit, or another special method (handled in the actions *Register*, *TakeLock*, *ReleaseLock*, etc.), *HandleSpecialMethodsMainLoop* recurses to allow further special methods to be handled. The return from the top-level method execution on the *main* thread is only permitted once all memory areas have been exited and *memoryEntries* is zero for all threads.

To illustrate how entering memory areas operates, we show the *ExecuteInAreaOf* action below, which is one of the actions offered in external choice in *EnterMemory*, along with actions to handle other memory-area entering operations. *ExecuteInAreaOf* takes a thread identifier *thread* as a parameter and only accepts communications from that thread, so that we can separate out memory-area entries for each thread. Such an identifier is received for all of the memory-area entering methods. In the case of *ExecuteInAreaOf*, another identifier, *object*, is also received and a *FindBackingStore* action is used to communicate with the memory manager to determine its backing store. This backing store is then entered, via communication on the *enterBackingStore* channel. The completion of the memory-area entering operation is then signalled to the *Interpreter* via the *executeInAreaOfRet* channel, so that the `run()` method of the `Runnable` object can be executed.

$$
\begin{array}{l}
\mathit{ExecuteInAreaOf} \mathrel{\widehat=} \mathbf{val}\, \mathit{thread} : \mathit{ThreadID} \bullet \\
\quad \mathbf{var}\, \mathit{bs} : \mathit{BackingStoreID};\ \mathit{object} : \mathit{ObjectID} \bullet \\
\quad \mathit{executeInAreaOf}?t : (t = \mathit{thread})?\mathit{obj} \longrightarrow \mathit{object} := \mathit{obj}; \\
\quad \mathit{FindBackingStore}(\mathit{object}, \mathit{bs}); \\
\quad \mathit{enterBackingStore}!\mathit{thread}!\mathit{bs} \longrightarrow \mathit{executeInAreaOfRet} \longrightarrow \mathbf{Skip}
\end{array}
$$

After the `run()` method execution has finished, a further special method is called in the *Interpreter* to exit the memory area. Its handling is specified by the *ExitMemory* action below. This, as with the *ExecuteInAreaOf* action, takes a *thread* parameter. A return from a method executing on that thread is accepted on the *exitMemory* channel. The exit from the memory area is then triggered using the *exitBackingStore* and *exitBackingStoreRet* channels. The *Launcher* state may afterwards be updated to account for the exited memory area being cleared (due to no longer being in use by any thread), which is specified in the *ClearPrivateMemory* schema. After the exit from the memory area has been handled, the *Launcher* signals that the special method handling has finished using the *exitMemoryRet* channel.

$$
\begin{array}{l}
\mathit{ExitMemory} \mathrel{\widehat=} \mathbf{val}\, \mathit{thread} : \mathit{ThreadID} \bullet \\
\quad \mathit{exitMemory}?t : (t = \mathit{thread}) \\
\quad \longrightarrow \mathit{exitBackingStore}!\mathit{thread} \longrightarrow \mathit{exitBackingStoreRet}?\mathit{bsid}?\mathit{isCleared} \longrightarrow \\
\quad \mathbf{if}\ \mathit{isCleared} = \mathbf{True} \longrightarrow \big(\mathit{ClearPrivateMemory}[\mathit{bsid}/\mathit{toClear}?]\big) \\
\quad [\!]\ \mathit{isCleared} = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\quad \mathbf{fi}\,;\ \mathit{exitMemoryRet} \longrightarrow \mathbf{Skip}
\end{array}
$$

This handling of special methods is used by the interpreter (or C program, after compilation), which communicates with the *Launcher* when such methods are encountered. We describe in detail how this communication is performed in the interpreter in Section 4.3.4, and in the C program in Section 4.4.1.

During the execution of a mission in *WaitForExecution*, the release of the event handlers of the mission must also be managed by the *Launcher*. This is performed by the *ExecuteThreads* action, shown below, which is executed in parallel interleaving with the action above to handle special methods. In this interleaving, the state is partitioned such that the *AlarmManager*, the *SchedulableManager* and the *perReleaseMemories* map are controlled by *ExecuteThreads*, with all other state components controlled by the special-method handling action. *ExecuteThreads* takes a parameter, *missionStart*, which is the start time of the mission received from the real-time clock in *WaitForExecution*.

$$
\begin{array}{l}
\mathit{ExecuteThreads} \mathrel{\widehat=} \mathbf{val}\, \mathit{missionStart} : \mathit{Time} \bullet \\
\quad (\mathit{HandleAlarms} \\
\quad [\![\ \{\mathit{alarms}\} \mid \{\!|\ \mathit{setAlarm}, \mathit{triggerAlarm}\ |\!\} \mid \varnothing\ ]\!] \\
\quad (\big|\big|\big|\ \mathit{obj} : \mathit{ObjectID} \setminus \{\mathit{clockHandler}\}\ |\!|\![\ \varnothing]\!|\!|\ \bullet \mathit{HandleThread}(\mathit{obj}, \mathit{missionStart}))) \\
\quad \setminus \{\!|\ \mathit{setAlarm}, \mathit{triggerAlarm}\ |\!\}
\end{array}
$$

ExecuteThreads is itself a parallelism of actions: *HandleAlarms*, which manages the setting of alarms for the schedulable objects of the mission, and *HandleThread*, which manages the release of the schedulable objects on their threads. Information is transferred to and from *HandleAlarms* via the channels *setAlarm* and *triggerAlarm*, which are hidden. A separate copy of *HandleThread* manages each possible schedulable object, with the exception of *clockHandler*, which is managed by *HandleAlarms*.

The *HandleAlarms* action, shown below, offers a choice of accepting a request to set a new alarm for a schedulable object's thread via the *setAlarm* channel, and accepting the execution of the *clockHandler* interrupt thread via *runThread*. When a request to set an alarm is accepted, the thread's identifier, *thread*, and the time of the desired alarm, *alarmTime*, are received via *setAlarm*. These are then added to the *alarms* map with *thread* associated to *alarmTime*, in the data operation *AddAlarm*. Execution of the *clockHandler* thread begins with receiving its identifier, *tid*, an object identifier, which must be that of *clockHandler*, and a method identifier, that of the `handle()` method. The threads on which an alarm is due to trigger next are then removed from the domain of *alarms* in the data operation *PopAlarm*, and returned as a set *threads*. Each of the thread's whose identifier is in the *threads* set are then signalled via the *triggerAlarm* channel. The *clockHandler* thread execution is then signalled to finish via the *endThread* channel so that the interrupt handler can execute again.

$$
\begin{array}{l}
\mathit{HandleAlarms} \mathrel{\widehat=} \mathbf{var}\ \mathit{updateAlarm} : \mathbb{B} \bullet \\
\quad \left(\begin{array}{l}
\left(\mathit{setAlarm}?\mathit{thread}?\mathit{alarmTime} \longrightarrow \left(\mathit{AddAlarm}\right) \right) \\
\Box \\
\left(\begin{array}{l}
\mathit{runThread}?\mathit{tid}?\mathit{oid} : (\mathit{oid} = \mathit{clockHandler})?\mathit{mid} : (\mathit{mid} = \mathit{handle}) \\
\ \longrightarrow \mathbf{var}\ \mathit{threads} : \mathbb{F}\ \mathit{ThreadID} \bullet \left(\mathit{PopAlarm}\right); \\
(\text{;}\ \ \mathit{thread} : \mathit{threads} \bullet \mathit{triggerAlarm}!\mathit{thread} \longrightarrow \mathbf{Skip}); \\
\mathit{endThread}!\mathit{tid} \longrightarrow \mathbf{Skip}
\end{array} \right)
\end{array} \right); \\
\quad \mu X \bullet \\
\qquad \mathbf{if}\ \mathit{updateAlarm} = \mathbf{True} \longrightarrow \\
\qquad\qquad \mathbf{var}\ \mathit{nextAlarm} : \mathit{Time} \bullet \left(\mathit{GetNextAlarm}\right); \\
\qquad\qquad \mathit{RTCsetAlarm}!\mathit{nextAlarm} \longrightarrow \mathit{RTCreport}?\mathit{report} \longrightarrow \\
\qquad\qquad \mathbf{if}\ \mathit{report} = \mathit{RTCokay} \longrightarrow \mathbf{Skip} \\
\qquad\qquad\ [\!]\ \mathit{report} = \mathit{RTCtimeInPast} \longrightarrow \\
\qquad\qquad\qquad \mathbf{var}\ \mathit{threads} : \mathbb{F}\ \mathit{ThreadID} \bullet \left(\mathit{PopAlarm}\right); \\
\qquad\qquad\qquad (\text{;}\ \ \mathit{thread} : \mathit{threads} \bullet \mathit{triggerAlarm}!\mathit{thread} \longrightarrow \mathbf{Skip})\ ;\ X \\
\qquad\qquad \mathbf{fi} \\
\qquad\ [\!]\ \mathit{updateAlarm} = \mathbf{False} \longrightarrow \mathbf{Skip} \\
\qquad \mathbf{fi}\ ;\ \mathit{HandleAlarms}
\end{array}
$$

After both of the actions offered by *HandleAlarms*, the alarm set in the real-time clock may need updating. This is indicated by a boolean variable *updateAlarm*, which is set by *AddAlarm* and *PopAlarm*. If *updateAlarm* is **True**, the time of the next alarm in *alarms* is determined by *GetNextAlarm*. The alarm is then set in the real-time clock via the *RTCsetAlarm* channel and the report returned via the *RTCreport* channel is checked. If the alarm could not be set due to the alarm time having already passed, then the alarm is triggered immediately, as in the case of the *clockHandler* thread executing, and *updateAlarm* is checked again. If the alarm is successfully set in the real-time clock or there is no alarm to set, then *HandleAlarms* recurses to offer the choice of actions again.

The *HandleThread* action, which is executed for each object in parallel with *HandleAlarms*, is composed of a choice of actions to handle each type of event handler. As mentioned previously, we handle only aperiodic, periodic and one-shot handlers without deadline miss handlers. Extensions to provide for additional event handler types would only require modifying the actions of this choice, and so would not affect the compilation strategy as the change would be restricted to the *Launcher*.

As an example of one of the actions in this choice, we present the *OneShotEventHandlerRel* action, which models the execution of a one-shot event handler that is initialised with a relative release time. It takes an object identifier *oseh* as a parameter, along with the *missionStart* time. These are the same as the parameters passed to *HandleThread*. The action is guarded so that it is only offered if *oseh* is a registered schedulable object with a type of *OneShotRel*.

The relative time offset, *startTime*, is extracted from *schedulableTypes* for *oseh* and the alarm is set by communicating with *HandleAlarms* on the *setAlarm* channel. The alarm time is computed by adding *startTime* to *missionStart*, since *startTime* is a relative offset from the time the mission starts. This is performed before execution of the thread begins to ensure that the alarms for all the threads are set when the mission starts. The thread for the alarm is that associated with *oseh* in *schedulableThreads*. The execution of the thread itself occurs in *OneShotHandlerExecution*, which is used for one-shot handlers with both relative and absolute release times.

$$
\begin{array}{l}
\mathit{OneShotEventHandlerRel} \mathrel{\widehat=} \mathbf{val}\, \mathit{oseh} : \mathit{ObjectID};\ \mathbf{val}\, \mathit{missionStart} : \mathit{Time} \bullet \\
\quad (\mathit{oseh} \in \mathit{registeredObjects} \land \mathit{schedulableTypes}\ \mathit{oseh} \in \operatorname{ran} \mathit{OneShotRel}) \& \\
\quad \mathbf{var}\, \mathit{startTime} : \mathit{Time} \bullet \mathit{startTime} := ((\mathit{OneShotRel}^{\sim})\ (\mathit{schedulableTypes}\ \mathit{oseh})); \\
\quad \mathit{setAlarm}!(\mathit{schedulableThreads}\ \mathit{oseh})!(\mathit{missionStart} + \mathit{startTime}) \\
\quad\ \longrightarrow \mathit{OneShotHandlerExecution}(\mathit{oseh})
\end{array}
$$

OneShotHandlerExecution, shown below, takes the *oseh* object identifier passed to it from *OneShotEventHandlerRel* as a parameter. It accepts communication on the *runThread* channel, receiving a thread, which is required to be the same as that associated with *oseh* in *schedulableThreads*, an object identifier, which is required to be the same as *oseh*, and a method identifier. The thread identifier is stored as *thread* for use later in this action. After a communication on *runThread* has been accepted, the thread suspends itself to allow other threads to run while it waits for its release. When it receives a signal on *triggerAlarm* for *thread*, then it signals to the scheduler to resume its thread so that it can execute methods. We check that *thread* has an associated backing store in *perReleaseMemories*, diverging if no such backing store exists. If such a backing store does exist, then we enter it and trigger the execution of the `handleAsyncEvent()` method of the object *oseh* using the *executeMethod* channel. After the method returns, the backing store is exited using the *exitBackingStore* channel and the end of the thread's execution is signalled via the *endThread* channel, since there are no further releases of the event handler. We do not consider the rescheduling of one-shot event handlers here, but it could be handled by having the rescheduling method as a special method.

```
$OneShotHandlerExecution \mathrel{\widehat=} \mathbf{val}\, oseh : ObjectID \bullet \mathbf{var}\, thread : ThreadID \bullet$
    $runThread?t : (t = schedulableThreads\ oseh)?obj : (obj = oseh)?m \longrightarrow thread := t;$
    $SuspendThread\ ;\ triggerAlarm?t : (t = thread) \longrightarrow ResumeThread(thread);$
    $\mathbf{if}\ thread \in \operatorname{dom} perReleaseMemories \longrightarrow$
        $enterBackingStore!thread!(perReleaseMemories\ thread)$
        $\longrightarrow getClassIDOf!oseh?cid$
        $\longrightarrow executeMethod!thread!cid!handleAsyncEvent!(\langle oseh \rangle)$
        $\longrightarrow executeMethodRet?void$
        $\longrightarrow exitBackingStore!thread \longrightarrow endThread!thread \longrightarrow \mathbf{Skip}$
    $[\!]\ thread \notin \operatorname{dom} perReleaseMemories \longrightarrow \mathbf{Chaos}$
    $\mathbf{fi}$
```

In the next section, we describe the bytecode interpreter, which, along with the *Launcher*, forms the CEE before the application of the compilation strategy.

4.3 Bytecode Interpreter Model

This section describes the bytecode interpreter that handles execution of an SCJ bytecode program. Its model is composed of two processes: the model of the object manager, *ObjMan*, and the model of the interpreter itself, *Interpreter*. These are composed together in parallel with the *Launcher* to form the complete core execution environment, *CEE*, as shown below. The synchronisation sets and channel hidings, omitted here, are consistent with the communication patterns shown in Table 4.1.

$$\begin{aligned}&CEE(cs, bc, instCS, sid, initOrder) \mathrel{\widehat=}\\&\quad ObjMan(cs) \parallel Interpreter(cs, bc, instCS) \parallel Launcher(sid, initOrder)\end{aligned}$$

CEE is parametrised by values that characterise a particular program: *bc*, recording the bytecode instructions, *cs*, recording information about the classes in the program, *instCS*, recording the classes that are instantiated in the program, *sid*, recording the identifier of the `Safelet` class, and *initOrder*, a sequence of class identifiers indicating in which order the classes should be initialised. These parameters are passed to the components that use them. The *sid* and *initOrder* parameters have been explained in Section 4.2. We describe the *cs*, *bc* and *instCS* later in this section where the processes that use them are described.

ObjMan manages the cooperation between the SCJ program and the SCJVM memory manager. This includes the representation of objects, since the SCJVM memory manager is agnostic as to the structure of objects and objects are shared between threads of the program. *ObjMan* also tracks the current memory area for each thread so that objects can be allocated in the correct memory area.

Interpreter and *Launcher* define the control flow and semantics of the SCJ program. The interpreter is for a representative subset of Java bytecode that covers stack manipulation, arithmetic, local variable manipulation, field manipulation, object creation, method invocation and return, and branching. This covers the main concepts of Java bytecode. We do not include instructions for different types as that would add duplication to the model while yielding no additional verification power. We also do not include exception handling as SCJ programs can be statically verified to prove that exceptions are not thrown [52, 72]. Furthermore, reliance on exceptions to handle errors has been discouraged by an empirical study due to the potential for errors in exception handling [103]. Errors caused in the SCJVM by an incorrect input program are represented by abortion.

Within *Interpreter*, there is one process for each thread identifier in the set of possible thread identifiers, with the exception of the *idle* thread, which performs no execution. Each of these processes represents an interpreter for a separate thread, with thread switches coordinated by communication between threads. They begin with state initialisation, followed by a choice of separate behaviours for the *main* thread and all other threads. The *main* thread offers a choice of executing a method in response to a signal from the *Launcher* or switching to another thread. The other threads wait for a signal from the scheduler instructing them to start execution, after which they wait for the scheduler to indicate they have been switched to and proceed to execute a method.

The execution of a method is performed in the same way for all threads, repeatedly handling individual bytecode instructions until all methods on the call stack have been returned from. Inbetween bytecode instructions, the scheduler is polled to check for thread switches. After method execution has finished, the *Launcher* and the scheduler are signalled as appropriate, and the thread's process returns to the start of its behaviour.

Next, in Section 4.3.1, we give an informal description of the bytecode instructions handled in our model and the ways in which their SCJ semantics differ from that of standard Java. In Section 4.3.2, we describe our model of Java class information that is used by both *ObjMan* and *Interpreter*. The *ObjMan* component is then described in Section 4.3.3 and *Interpreter* is described in Section 4.3.4.

4.3.1 Bytecode Subset

We model a subset of Java bytecode sufficient to express a wide variety of SCJ programs and illustrate how further features may be added. Additional instructions would be similar to those already defined, and so would add little value to establishing the scientific basis of our work. The semantics of any additional instructions and the compilation rules required for them can be obtained from the semantics of the instructions in our subset and the compilation rules defined in Chapter 5. Indeed, our prototype implementation of our compilation strategy, described in Section 6.3, implements some additional instructions to support the examples described in Section 6.4. For example, many Java bytecode instructions differ only in the types they operate over, so such instructions are handled in the same way by our compilation strategy.

The subset has been chosen by considering the bytecode generated from a simple SCJ program and removing instructions similar to those already in the subset. This ensures the model is not unnecessarily complicated with trivial or redundant instructions, so we can concentrate on the instructions that are most of interest in creating the compilation strategy. The bytecode instructions in our subset are described in Table 4.2.

Java bytecode instructions operate over a state that records information on all loaded classes, a stack frame, and the object data residing in memory. Various pieces of class information are required for execution of bytecode instructions, but a constant pool, which stores all the constants and names required by the class, is the main information used.

The constant pool contains references to classes, methods and fields used by the bytecode

Instruction	Parameter	Description
`aconst_null`	(none)	Pushes a null object reference onto the operand stack.
`aload`	local variable index	Loads the value from a specified local variable and pushes it onto the operand stack.
`areturn`	(none)	Returns from the current method, pushing the value on top of the current method's operand stack onto the operand stack of the method returned to.
`astore`	local variable index	Pops a value from the operand stack and stores it in the specified local variable.
`dup`	(none)	Duplicates the value on top of the operand stack.
`getfield`	constant pool index	Pops an object reference from the operand stack, gets the value of the field specified by the identifier at the given constant pool index for the referenced object, and pushes it onto the operand stack.
`getstatic`	constant pool index	Gets the value of the static field specified by the field and class identifiers at the given constant pool index, and pushes it onto the operand stack.
`goto`	program address offset	Unconditionally branches to the given program address.
`iadd`	(none)	Pops two integer values from the operand stack, adds them, and pushes the result onto the operand stack.
`iconst`	integer value	Pushes the given integer value onto the operand stack of the current method.
`if_icmple`	program address offset	Pops two integer values from the operand stack, and branches to the given program address if the second value popped is less than or equal to the first value.
`ineg`	(none)	Pops an integer value from the operand stack, negates it, and pushes the negated value onto the operand stack.
`invokespecial`	constant pool index	Gets the method and class identifier at the given constant pool index and invokes the specified method of the specified class (or, if the specified class is a superclass of the current class and the method is not a constructor, the direct superclass of the current class), popping the method's arguments, including a `this` object reference, from the operand stack.

continued on next page

Instruction	Parameter	Description
`invokestatic`	constant pool index	Gets the method and class identifier at the given constant pool index and invokes the specified static method of the specified class, popping the method's arguments from the operand stack.
`invokevirtual`	constant pool index	Gets the method and class identifier at the given constant pool index, pops the arguments of the specified method, including a `this` object reference, from the operand stack, and invokes the specified method of the class of the referenced object.
`new`	constant pool index	Allocates a new object of the class specified by the identifier at the given constant pool index and pushes a reference to the new object onto the operand stack.
`putfield`	constant pool index	Pops an object reference and value from the operand stack and stores the value in the field specified by the identifier at the given constant pool index for the referenced object.
`putstatic`	constant pool index	Pops a value from the operand stack and stores the value in the static field specified by the field and class identifiers at the given constant pool index.
`return`	(none)	Returns from a method with no return value.

Table 4.2: The instructions in our bytecode subset

instructions in the class, as well as constant values used in the code. The form of the constant pool is a large array. Indices into this array are used as parameters to instructions requiring information from the constant pool. For example, the `getfield` and `putfield` instructions take constant pool index parameters pointing to a reference to a field whose value should be obtained or set. Other class information used at runtime includes information on fields and methods belonging to the class, which is required for creation of objects and invocation of methods.

The frame stack forms the second part of the JVM manipulated by bytecode instructions and consists of a series of frames that contain the runtime information for each invocation of a method. When a method is invoked, a new stack frame is created for it and pushed onto the frame stack, and when the method returns, the stack frame is popped from the stack.

Each stack frame contains an operand stack, which is used to store values manipulated by bytecode instructions, and an array of local variables. Most bytecode instructions manipulate the operand stack in some way, popping arguments from it, pushing results to it or performing specific operations upon it.

The local variables are used to store the arguments of a method and the results of computations performed on the operand stack. Operations are not performed directly on the local variables, so the only bytecode instructions that affect them are those for moving values between the operand stack and the local variables (`aload` and `astore` are examples of such instructions).

Some bytecode instructions also manipulate objects, which in our case reside in backing store memory. Such instructions include `new`, which creates objects, and `getfield`, which gets the value from a field of an object. In our choice of instructions for the subset, we mainly focus on manipulation of objects and method invocation, since those are core concepts of Java bytecode and require special handling by the compilation strategy.

The instruction `dup` is included as an example of a simple instruction that operates on the operand stack. It has been chosen for its frequent occurrence in object initialisation. Other instructions that do simple operand stack manipulation, including the arithmetic instructions, can be specified similarly.

We also include a few arithmetic instructions as an example of how integers are handled. Specifically, we include the integer addition operation, `iadd`, as an example of a binary operation, and the integer negation operation, `ineg`, as an example of an unary operation. We do not include operations for floating point values since the operations upon them are not substantially different from those on integers at the level of modelling and compilation. The model can be easily extended to include more integer operations.

Instructions that create object references (the `new` and `aconst_null` instructions), pass them around (`aload`, `astore`, `areturn`, etc.), and permit field accesses (`getfield` and `putfield`) are also included to allow the full range of object manipulations. We also provide instructions for `static` field accesses (`getstatic` and `putstatic`) since they are of use in sharing data between different parts of the program. However, arrays are not included as they require additional instructions and can be emulated, albeit inefficiently, with the instructions given here.

Both the `invokevirtual` and `invokespecial` instructions, which invoke methods on objects, are included. The `invokevirtual` instruction looks up the method to invoke in the method table for the class of the object that the method is invoked on. The `invokespecial` instruction, on the other hand, uses the class identifier supplied in the method reference pointed to by the parameter of the instruction when looking up the method. The `invokestatic` instruction, for invoking `static` methods of classes, is similar to `invokespecial`, but does not supply a `this` object parameter, whereas `invokevirtual` and `invokespecial` pop `this` from the stack as an extra argument.

The `goto` and `if_icmple` instructions are provided as examples of control flow instructions, with `goto` representing an unconditional branch and `if_icmple` representing a conditional branch. Other forms of conditional branch may be implemented in a similar fashion to `if_icmple`, but we do not include those in our subset since `if_icmple` is sufficient to represent most control flow structures. Although `goto` could be represented as a special case of `if_icmple`, we include it as a separate instruction due to its frequent use in conjunction with `if_icmple` to implement loops.

We do not handle exceptions; errors in the SCJVM are instead handled by simply aborting execution. SCJ programs can be statically verified to prove that exceptions will not be thrown [52, 72]. Furthermore, reliance on exceptions to handle errors has been discouraged by an empirical study due to the potential for errors in exception handling [103]. The bytecode instructions that relate to throwing and catching exceptions are, therefore, not included in our bytecode subset.

As a simplifying assumption, we consider that all values consist of only a single virtual machine word. This means that `long` and `double` values are not handled. The reason for this assumption is that handling of two word values makes little difference at the level of the formal model and our approach can be easily extended to deal with more types.

Further, we do not make a distinction between the different virtual machine types in our bytecode instructions. This is justified as the bytecode instructions simply handle values as 32-bit words, with the type information only used for typechecking during bytecode validation. The code passed into the core execution environment is assumed to have already passed bytecode verification, which may have been done by a separate component [29, 53, 58, 112]. Since many of the instructions behave the same for different types, we only include those instructions that handle values as object references. We would introduce a lot of duplication in the model if, for example, both the `areturn` and `ireturn` instructions were to be included.

Because we are considering bytecode arising from an SCJ program, some requirements of SCJ permit further simplifications to our bytecode subset. The `invokedynamic` instruction performs method invocation with runtime typechecking, mainly for the purpose of implementing dynamically-typed languages targeting the JVM (though it is also used to implement the lambda expressions introduced in Java 8). It is not included in our subset as it does not allow static typechecking and so should not be used for SCJ.

The requirement for all classes to be loaded at startup greatly simplifies the semantics of several instructions, since dynamic class loading does not need to be considered. It also means that many run-time errors pertaining to method and field resolution can checked ahead-of-time to make sure they are not thrown. This means checks to make sure methods and fields exist, and that they have the correct access modifiers, do not need to be included in the run-time semantics of the instructions in our subset.

The `invokevirtual` and `invokeinterface` instructions exist as separate bytecodes in order to facilitate efficient method dispatch using method lookup tables. For methods defined in classes, invoked by `invokevirtual`, the methods of each class can simply be appended to the lookup table of its superclass, since there is a linear inheritance hierarchy. Methods defined in interfaces can be inserted at any point in the inheritance hierarchy, and so must be defined in separate method tables, using an approach such as that described in [2]. The separate `invokevirtual` and `invokeinterface` instructions allow implementations to easily determine which method table should be used.

However, the JVM specification does not require the use of method tables to implement method lookup so the semantics of the `invokevirtual` and `invokeinterface` instructions given in the JVM specification are actually quite similar, differing only in the types of methods they operate over, which can be checked ahead-of-time when we have all classes available. The `invokevirtual` instruction also provides for special methods called *signature-polymorphic methods*, but these form part of the infrastructure for the `invokedynamic` instruction and are not included in SCJ, so we do not include handling of them in our semantics. Given these considerations and the fact that our model of an SCJVM is a specification that does not require any specific implementation, such as method tables, we treat `invokevirtual` and `invokeinterface` the same and only include `invokevirtual` in our subset.

In terms of concurrency considerations, we are assuming our SCJVM to be single processor, and so we do not need to have more than one interpreter. As we see later, the interpreter's threads are modelled using separate *Circus* processes, but execution only occurs on one at a time. We also assume that thread switches can only occur between bytecode instructions in the interpreter. This is justified since bytecode instructions should appear to be atomic. An implementation may be non-atomic as long as the externally visible sequence of events is the same as for the model with atomic instructions. This means that instructions requiring communication with other components of the SCJVM, such as `new`, which communicates with the memory manager, must be atomic since they affect shared state.

Having described our bytecode subset and the assumptions we are making, we now proceed to describe our model of Java classes in the next section.

4.3.2 Classes

In our model, information about the Java classes that form the program is recorded in a map, *cs*, that is provided as a parameter to *CEE*. The *cs* map associates *ClassID*s with records of a schema type *Class* defined as the conjunction of three schemas. The first schema, *ClassConstantPool* contains components that represent the constant pool and indices into the constant pool. The second schema, *ClassMethods*, represents information on the methods in the class. The final schema, *ClassFields*, is our model for information on the fields in the class.

The components of *ClassConstantPool* are *constantPool*, the constant pool itself, and some indices into *constantPool*: *this*, referencing the current class, *super*, referencing the current class' superclass, and *interfaces*, a set of indices referencing the interfaces implemented by the current class.

```
ClassConstantPool
  constantPool : CPIndex ⇸ CPEntry
  this, super : CPIndex
  interfaces : 𝔽 CPIndex
  ──────────
  nullCPIndex ∉ dom constantPool
  {this} ∪ ({super} \ {nullCPIndex}) ∪ interfaces ⊆ dom constantPool
  constantPool ⦇ {this, super} ∪ interfaces ⦈ ⊆ ran ClassRef
```

The entries of *constantPool* are indexed by elements of a type *CPIndex*. In the JVM, the *CPIndex* values are positive integers, but no arithmetic or comparison is performed on constant pool indices in our model, so we do not represent that fact.

We distinguish one particular *CPIndex* value, a constant *nullCPIndex*, which represents an invalid index into the *constantPool*. It is used as a placeholder in cases when no index is present. For example, the class `Object` has no superclass, so the index of the constant pool entry referencing its superclass is *nullCPIndex*.

Each of the entries in the *constantPool* is represented by an element of a free type *CPEntry*, the definition of which is shown below. It has three constructors: *ClassRef*, representing a reference to a *ClassID*, *MethodRef*, representing a reference to a method of a particular class by a *ClassID* and *MethodID*, and *FieldRef*, representing a reference to a field of a particular class by a *ClassID* and *FieldID*.

$$
\begin{aligned}
&\mathit{CPEntry} ::= \mathit{ClassRef}\langle\!\langle \mathit{ClassID} \rangle\!\rangle \\
&\qquad \mid \mathit{MethodRef}\langle\!\langle \mathit{ClassID} \times \mathit{MethodID} \rangle\!\rangle \\
&\qquad \mid \mathit{FieldRef}\langle\!\langle \mathit{ClassID} \times \mathit{FieldID} \rangle\!\rangle
\end{aligned}
$$

Although there are other types of constant pool entry described in the JVM specification, we do not include them in our model since some of them are not relevant to our subset. Some constant pool entries are used by other constant pool entries. For example, in the JVM specification, method references reference another constant pool entry, which in turn contains references to further constant pool entries with string representations of the method's name and type. In our model, we hide this complexity in the identifier types *ClassID*, *MethodID* and *FieldID*, omitting the extra constant pool entries.

The first conjunct of the invariant of *ClassConstantPool* requires that *nullCPIndex* not be in the domain of *constantPool*, since *nullCPIndex* is not a valid index into *constantPool*. The second conjunct states that the indices *this*, *super*, and *interfaces* must be in the domain of *constantPool*, unless *super* is *nullCPIndex* (which is the case for the `Object` class). Finally, the third conjunct requires that the *constantPool* entries at *this*, *super* and *interfaces* are *ClassRef*s.

The components of *ClassMethods*, shown below, are maps from *MethodID* values to information about each method. The first two, *methodEntry* and *methodEnd*, map to *ProgramAddress* values, which are indices into a separate bytecode array representing the start and end points of the method. The next two components, *methodLocals* and *methodStackSize*, map to natural numbers giving the required number of local variables and operand stack slots for the method. These values are used during the compilation strategy to declare C variables to store the local variables and operand stack values. The final two components, *staticMethods* and *synchronizedMethods*, are sets of *MethodID* values containing the static and synchronized methods of the class respectively. The methods not in the *staticMethods* are considered to be non-static methods and must take an additional `this` argument.

ClassMethods

$$\begin{aligned}
&\mathit{methodEntry}, \mathit{methodEnd} : \mathit{MethodID} \pfun \mathit{ProgramAddress}\\
&\mathit{methodLocals}, \mathit{methodStackSize} : \mathit{MethodID} \pfun \mathbb{N}\\
&\mathit{staticMethods}, \mathit{synchronizedMethods} : \mathbb{F}\, \mathit{MethodID}
\end{aligned}$$

$$\begin{aligned}
&\operatorname{dom} \mathit{methodEntry} = \operatorname{dom} \mathit{methodEnd}\\
&\qquad = \operatorname{dom} \mathit{methodLocals} = \operatorname{dom} \mathit{methodStackSize}\\
&\mathit{staticMethods} \subseteq \operatorname{dom} \mathit{methodEntry}\\
&\mathit{synchronizedMethods} \subseteq \operatorname{dom} \mathit{methodEntry}\\
&\forall\, m : \operatorname{dom} \mathit{methodEntry} \bullet \mathit{methodEntry}\ m \leq \mathit{methodEnd}\ m\\
&\qquad \land (m \in \mathit{staticMethods} \Rightarrow \mathit{methodArguments}\ m \leq \mathit{methodLocals}\ m)\\
&\qquad \land (m \notin \mathit{staticMethods} \Rightarrow \mathit{methodArguments}\ m + 1 \leq \mathit{methodLocals}\ m)
\end{aligned}$$

In addition to the components of *ClassMethods*, we declare a global function *methodArguments* from *MethodID*s to natural numbers, which gives the number of arguments that each method takes. This is a global function since each *MethodID* encodes the type of the method, so the number of arguments for a method can always be determined from its identifier. The *methodArguments* function is also total for this reason. We use *methodArguments* in the invariant of *ClassMethods*, and also in the *Interpreter* and compilation strategy when handling method calls.

The first conjunct of the invariant of *ClassMethods* requires that all the component maps have the same domain, so that all the information must be supplied for any method present in the class. The second conjunct requires that every *MethodID* in *staticMethods* be within the domains of these maps. The third conjunct states a similar requirement for *synchronizedMethods*. Finally, the fourth conjunct requires that, for each method, its *methodEntry* is before its *methodEnd*, and that its *methodLocals* is large enough to contain its *methodArguments*, plus an extra `this` argument for non-static methods, since each argument of a method is stored in a local variable.

The final components of our model for class information are given in the *ClassFields* schema below. It contains two sets of *FieldID*s, *fields* and *staticFields*, which are the identifiers of the class' object fields and static fields respectively. The static and non-static fields need to be distinguished so that we know where each needs to be stored: static variables have only one copy for each class, whereas non-static fields are stored separately for each instance of a class. The *fields* and *staticFields* sets are required to be disjoint since no field can be both static and non-static.

$$\begin{array}{|l}
\hline
\textit{ClassFields} \\
\textit{fields}, \textit{staticFields} : \mathbb{F}\, \textit{FieldID} \\
\hline
\textit{fields} \cap \textit{staticFields} = \varnothing \\
\hline
\end{array}$$

The three schemas containing the different parts of the class information are conjoined together to form *Class*, as shown below.

$$\textit{Class} == \textit{ClassConstantPool} \land \textit{ClassMethods} \land \textit{ClassFields}$$

In addition to defining *Class*, we also define functions for extracting information from the *constantPool* for a given *Class*, in order to make specifying things about them easier. Since the functions are just abbreviations of data access operations, we omit them here. We recall that the definitions omitted here are given in Appendix B of the extended version of this thesis [13].

We also require a way of expressing the fact that one class is a subclass of another (or implements a given interface). We say that a *Class* binding, $c1$, is a direct subclass of another class, $c2$, written $c1 \prec_{\mathrm{d}} c2$, if the *this* identifier of $c2$ is the *super* identifier of $c1$ or one of its *interfaces* identifiers.

We also define a relation, *subClassRel*, between class identifiers $cid1$ and $cid2$ in terms of the $\prec_{\mathrm{d}}$ relation. This requires a map from *ClassID* to bindings of *Class*, which is provided as a parameter to *subclassRel*. Given such a map, *cs*, we define $(cid1, cid2) \in subclassRel\ cs$ to hold if, in *cs*, $cid1$ and $cid2$ refer to *Class* bindings such that $cs\ cid1 \prec_{\mathrm{d}} cs\ cid2$ holds. We expand *subclassRel* to refer to its reflexive transitive closure so that it includes indirect subclass relationships and classes being subclasses of themselves. We omit the formal definitions of $\prec_{\mathrm{d}}$ and *subclassRel* here.

The *cs* map provided as a parameter to *CEE* is used as the parameter to *subclassRel* in each of the processes that uses it. In order for the CEE to execute the program, this *cs* parameter must represent a valid SCJ program, with all the necessary classes present. If this holds, then *subclassRel* represents the usual notion of when an object of a Java class is assignable to a variable of a given class.

We next describe the object manager process, *ObjMan*, which uses the *Class* type and the *cs* map.

4.3.3 Object Manager

The object manager, which is represented by the process *ObjMan*, manages the objects of the SCJ program executed by the core execution environment. This component is necessary because the SCJVM memory manager is agnostic to the structure of objects, which depends on the contents of the classes supplied as part of an SCJ program. Besides managing the creation and manipulation of objects, the object manager tracks the current allocation context for each thread. It ensures objects are allocated in the correct area, since the SCJVM memory manager is also agnostic to the existence of threads.

The start of the definition of *ObjMan* is shown below. *ObjMan* takes a single parameter, *cs*, which is a map from *ClassID*s to *Class* records containing the class information for each of the classes in the program. This information is used in determining the structure of the objects for each class.

$$\mathbf{process}\ \mathit{ObjMan} \mathrel{\widehat=} \mathit{cs} : \mathit{ClassID} \pfun \mathit{Class} \bullet \mathbf{begin}$$

Since the actual arrangement of an object in memory is an implementation consideration, the amount of memory required to store each object is implementation-defined. It is represented here by a global function *sizeOfObject*, declared below, which maps the information in each *Class* to the amount of memory required for objects of the class it represents.

$$\mathit{sizeOfObject} : \mathit{Class} \rightarrow \mathbb{N}$$

The objects that *ObjMan* operates on are described by records of the schema type *Object* shown below. It contains a map, *fields*, which associates the *FieldID* for each field of an object with a *Word* value stored in that field. A copy of the *Class* information for the object's class is also recorded in *class*. The invariant of *Object* requires that the domain of *fields* be the same as the fields given in *class*.

$$\begin{array}{|l}
\textit{Object} \\
\hline
\mathit{fields} : \mathit{FieldID} \pfun \mathit{Word} \\
\mathit{class} : \mathit{Class} \\
\hline
\operatorname{dom} \mathit{fields} = \mathit{class}.\mathit{fields} \\
\hline
\end{array}$$

The state of *ObjMan* is separated into three parts. The first part, *BackingStoreManager*, shown below, describes the layout of the backing stores used to store objects, since the memory manager is agnostic to objects and threads. Its first component, *backingStoreMap*, maps each *BackingStoreID* in use to an *ObjectID* set recording which objects are contained in that backing store. The second component, *backingStoreStacks*, maps *ThreadID*s to sequences of *BackingStoreID*s, recording which backing stores a thread has entered. The sequences must be non-empty since a thread that is in use must have a backing store associated with it. The third and final component of *BackingStoreManager* is *rootBS*, which stores the identifier of the root backing store.

$$\begin{array}{|l}
\textit{BackingStoreManager} \\
\hline
\mathit{backingStoreMap} : \mathit{BackingStoreID} \pfun \mathbb{F}\ \mathit{ObjectID} \\
\mathit{backingStoreStacks} : \mathit{ThreadID} \pfun \operatorname{seq}_1 \mathit{BackingStoreID} \\
\mathit{rootBS} : \mathit{BackingStoreID} \\
\hline
\operatorname{dom} \mathit{backingStoreMap} = \\
\qquad \bigcup \{ t : \operatorname{dom} \mathit{backingStoreStacks} \bullet \operatorname{ran} (\mathit{backingStoreStacks}\ t) \} \\
\mathit{rootBS} \in \operatorname{dom} \mathit{backingStoreMap} \\
\hline
\end{array}$$

The first conjunct of *BackingStoreManager*'s invariant requires the domain of *backingStoreMap* to be the identifiers of backing stores in *backingStoreStacks*, since a backing store that is not in use is cleared and so should not have any objects in it. The second conjunct requires that *rootBS* always be in the domain of *backingStoreMap*, since that represents immortal memory, which is never cleared.

The second part of the state for *ObjMan* is the schema *ObjectInfo*, shown below. This contains a map, *objects*, relating each *ObjectID* in use to the *Object* structure storing the information for the corresponding object. Its invariant relates the class information for each object to the classes in the *cs* parameter of *ObjMan*. It requires that, for each object *obj* in *objects*, the class identifier specified by the *this* field of the class information for *obj* must be the identifier of one of the classes in *cs*. The class information for *obj* must be the information for the corresponding class in *cs*, with its *fields* set replaced with the collected fields all superclasses of that class. This replacement of the fields in the invariant ensures field inheritance is properly handled.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ObjectInfo}} \\
\hline
\mathit{objects} : \mathit{ObjectID} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{Object} \\
\hline
\forall\, \mathit{obj} : \operatorname{ran} \mathit{objects} \bullet \exists\, \mathit{classID} : \mathit{ClassID} \bullet \\
\quad \mathit{classID} = (\mathit{thisClassID}\ \mathit{obj}.\mathit{class}) \land \mathit{classID} \in \operatorname{dom} \mathit{cs} \land \\
\quad \exists\, \Delta \mathit{Class} \mid (\Xi \mathit{Class}) \setminus (\mathit{fields}, \mathit{fields}') \bullet \\
\qquad \theta\, \mathit{Class} = \mathit{cs}\ \mathit{classID} \land \\
\qquad \mathit{fields}' = \\
\qquad\quad \bigcup \{ \mathit{cid} : \operatorname{dom} \mathit{cs} \mid (\mathit{classID}, \mathit{cid}) \in \mathit{subclassRel}\ \mathit{cs} \bullet (\mathit{cs}\ \mathit{cid}).\mathit{fields} \} \land \\
\qquad \mathit{obj}.\mathit{class} = \theta\, \mathit{Class}' \\
\hline
\end{array}
$$

The third part of *ObjMan*'s state is described by the schema *StaticFieldsInfo*, shown below, which stores information about the static fields of classes. Its component is the *staticClassFields* map, which relates pairs of class and field identifiers to *Word* values, representing the contents of each static field. The invariant of *StaticFieldsInfo* requires that *staticClassFields* contain every field specified by the *staticFields* component for each class in *cs*.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StaticFieldsInfo}} \\
\hline
\mathit{staticClassFields} : (\mathit{ClassID} \times \mathit{FieldID}) \mathrel{+\!\!\!\!\!\rightarrow} \mathit{Word} \\
\hline
\operatorname{dom} \mathit{staticClassFields} = \bigcup \{ \mathit{cid} : \operatorname{dom} \mathit{cs} \bullet \{\mathit{cid}\} \times (\mathit{cs}\ \mathit{cid}).\mathit{staticFields} \} \\
\hline
\end{array}
$$

These three parts are conjoined together to form the schema *ObjManState*, shown below, which is the state for *ObjMan*. This has an additional invariant, relating the parts together, which requires that the domain of *objects* is partitioned by the *ObjectID* sets given by *backingStoreMap*, so that every object is allocated in exactly one backing store.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ObjManState}} \\
\hline
\mathit{BackingStoreManager};\ \mathit{ObjectInfo};\ \mathit{StaticFieldsInfo} \\
\hline
\mathit{backingStoreMap} \mathrel{\mathrm{partition}} \operatorname{dom} \mathit{objects} \\
\hline
\end{array}
$$

The *ObjManState* is initialised as described in *ObjManInit* below. This operation takes the identifier of the root backing store as an input, *rootBS*?. The *objects* map is initialised to the empty set, since there are initially no objects in existence. The *backingStoreMap* initially contains only *rootBS*?, which is the only backing store initially in existence, associated with an empty set of object identifiers. The *backingStoreStacks* map is initialised to contain the *main* and *idle* threads, both with *rootBS*? as their only backing store entered. The *rootBS* identifier is set to be the same as the *rootBS*? input. Every static field in *staticClassFields* (whose domain is determined by the invariant of *StaticFieldsInfo*) is initially set to null. We do not consider the constant initial values provided for static fields in Java class files here, since they add little to the compilation strategy and can be emulated with class initialisers.

$$
\begin{array}{|l}
\textit{ObjManInit} \\
\hline
\textit{ObjManState}' \\
\textit{rootBS}? : \textit{BackingStoreID} \\
\hline
\textit{objects}' = \varnothing \\
\textit{backingStoreMap}' = \{\textit{rootBS}? \mapsto \varnothing\} \\
\textit{backingStoreStacks}' = \{\textit{main} \mapsto \langle \textit{rootBS}? \rangle, \textit{idle} \mapsto \langle \textit{rootBS}? \rangle\} \\
\textit{rootBS}' = \textit{rootBS}? \\
\forall x : \operatorname{dom} \textit{staticClassFields} \bullet \textit{staticClassFields}\ x = \textit{null} \\
\hline
\end{array}
$$

The *ObjMan* process then proceeds as described in its main action, shown below. It begins in *Init*, which communicates with the SCJVM memory manager to obtain the identifier of the root backing store and allocate space for *staticClassFields*, and then initialises the state as described in *ObjManInit*. Afterwards, in *Loop*, the process repeatedly offers each of its services in external choice.

$$\bullet\ \textit{Init} \ ; \ \textit{Loop}$$

After *ObjMan* is initialised, *ObjMan* offers services to the other components of the *CEE*, in the *Loop* action shown below. The services offered by *Loop* include *NewObject*, which creates an object of a given class. The *GetField* and *PutField* actions allow for obtaining and setting the value of an object's field. Similarly, *GetStatic* and *PutStatic* allow for obtaining and setting the value of a class' static fields. *GetClassIDOf* obtains the *ClassID* for the class of an object, by extracting the *this* identifier from the *Class* information for the object.

Management of allocation contexts is provided by the remaining services. The first service is *EnterBackingStore*, which enters a backing store for a given thread by pushing it onto the stack in *backingStoreStacks* for that thread, and adding it to *backingStoreMap* if it is not already in its domain. The second is the corresponding operation *ExitBackingStores* for exiting the current allocation context of a given thread, which means popping it from the thread's stack in *backingStoreStacks*, and clearing and removing the backing store from *backingStoreMap* if no threads are still using it. The *AddThreadMemory* and *RemoveThreadMemory* services allow for adding a thread to *backingStoreStacks* when it starts executing, and removing it when it finishes executing. Finally, the *GetCurrentAC* action obtains the current allocation context for a given thread, which is the backing store on top of its stack in *backingStoreStacks*.

$$
\begin{aligned}
\textit{Loop} \mathrel{\widehat=} (&\textit{NewObject} \mathbin{\square} \textit{GetField} \mathbin{\square} \textit{PutField} \mathbin{\square} \textit{GetStatic} \mathbin{\square} \textit{PutStatic} \mathbin{\square} \textit{GetClassIDOf} \\
&\mathbin{\square} \textit{EnterBackingStore} \mathbin{\square} \textit{ExitBackingStore} \mathbin{\square} \textit{AddThreadMemory} \\
&\mathbin{\square} \textit{RemoveThreadMemory} \mathbin{\square} \textit{GetCurrentAC}) \ ; \ \textit{Loop}
\end{aligned}
$$

The compilation refines the structure of objects. This means that field access operations (*GetField*, *PutField*) and object allocation (*NewObject*) are affected. *GetClassIDOf* is also affected since an object's class identifier is stored as part of its structure. We also refine the static fields data structure, requiring the operations upon it (*GetStatic*, *PutStatic*) to be changed. The management of allocation contexts is unaffected by the compilation, but is required for managing allocation of objects, so that they can be allocated in the correct backing store.

The definition of the *NewObject* action is shown below. It determines the *class* information for the new object using the data operation *GetObjectClassInfo*, which looks up the class identifier communicated on the *newObject* channel in the *cs* map, and replaces its *fields* with the union of its *fields* and those of its superclasses, to account for field inheritance. The space for the object is then allocated in the action *AllocateObject*, which communicates with the memory manager on the *MMallocateMemory* and *MMallocateMemoryRet* channels. The backing store used is the last backing store in *backingStoreStacks* for *thread* and the size required for the object is computed from the *class* information returned by *GetObjectClassInfo*, via the *sizeOfObject* function. The identifier of the new object is stored in *objectID*. After a successful allocation, the object is added to the *objects* map with its fields initialised to *null*, in *ObjManObjectInit*, and the object's identifier is returned via *newObjectRet*.

$$
\begin{array}{l}
\mathit{NewObject} \mathrel{\widehat=} \mathbf{var}\, \mathit{thread} : \mathit{ThreadID};\ \mathit{classID} : \mathit{ClassID} \bullet \\
\quad \mathbf{var}\, \mathit{objectID} : \mathit{ObjectID};\ \mathit{class} : \mathit{Class} \bullet \\
\quad \mathit{newObject}?t?c \longrightarrow \mathit{thread}, \mathit{classID} := t, c\ ;\ \big(\mathit{GetObjectClassInfo}\big); \\
\quad \mathit{AllocateObject}(\mathit{thread}, \mathit{sizeOfObject}\ \mathit{class}, \mathit{objectID}); \\
\quad \big(\mathit{ObjManObjectInit}\big)\ ;\ \mathit{newObjectRet}!\mathit{objectID} \longrightarrow \mathbf{Skip}
\end{array}
$$

We omit the definitions of the other actions of *Loop* here. The full model of the object manager can be found in Appendix B of the extended version of this thesis [13].

Next, we discuss the *Interpreter* process, which is the final component of *CEE* and handles the execution of the bytecode instructions themselves.

4.3.4 Interpreter

The *Interpreter* process is the final component of pre-compilation *CEE* that we present. It handles the execution of bytecode instructions: those in the subset described in Section 4.3.1, represented by the free type *Bytecode*, shown below. *Bytecode* has a constructor for each bytecode instruction, with any parameter to the instruction represented as a parameter of the constructor.

$$
\begin{array}{l}
\mathit{Bytecode} ::= \mathit{aconst\_null} \mid \mathit{dup} \mid \mathit{areturn} \mid \mathit{return} \mid \mathit{iadd} \mid \mathit{ineg} \\
\quad \mid \mathit{new}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \mid \mathit{iconst}\langle\!\langle \mathbb{N} \rangle\!\rangle \mid \mathit{aload}\langle\!\langle \mathbb{N} \rangle\!\rangle \mid \mathit{astore}\langle\!\langle \mathbb{N} \rangle\!\rangle \\
\quad \mid \mathit{getfield}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \mid \mathit{putfield}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \mid \mathit{getstatic}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \mid \mathit{putstatic}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \\
\quad \mid \mathit{invokespecial}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \mid \mathit{invokevirtual}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \mid \mathit{invokestatic}\langle\!\langle \mathit{CPIndex} \rangle\!\rangle \\
\quad \mid \mathit{if\_icmple}\langle\!\langle \mathbb{Z} \rangle\!\rangle \mid \mathit{goto}\langle\!\langle \mathbb{Z} \rangle\!\rangle
\end{array}
$$

The bytecode instructions are arranged in a map, *bc*, from *ProgramAddress* values (which are modeled by natural numbers) to *Bytecode* values. The *bc* map is passed as a parameter to the *Interpreter* process, along with the *cs* map described in Section 4.3.2, and a third parameter, *instCS*, which represents the set of class identifiers that are instantiated in the program. These parameters can be seen in the definition of *Interpreter* below.

The *instCS* set can be determined from the `new` instructions in *bc*, with the corresponding constant pool information in *cs*. This set determines the possible classes for an object, and hence what the possible targets for a virtual method call are. This is similar to the approach of icecap and allows the choice over targets of a method call, which compilation introduces, to be kept as small as possible.

The overall structure of *Interpreter* is a parallel composition of *Thr* processes representing the individual interpreter threads, with one process for each *ThreadID* except for *idle*. The *bc*, *cs* and *instCS* parameters are passed to each *Thr* process, along with its *ThreadID*.

$$
\begin{array}{l}
\mathbf{process}\ \mathit{Interpreter} \mathrel{\widehat=} \\
\quad \mathit{bc} : \mathit{ProgramAddress} \pfun \mathit{Bytecode};\ \mathit{cs} : \mathit{ClassID} \pfun \mathit{Class};\ \mathit{instCS} : \mathbb{F}\,\mathit{ClassID} \bullet \\
\quad \Vert\, t : \mathit{ThreadID} \setminus \{\mathit{idle}\} \ \llbracket\ \mathit{ThrChans}(t) \rrbracket \bullet \mathit{Thr}(\mathit{bc}, \mathit{cs}, \mathit{instCS}, t)
\end{array}
$$

Each $\mathit{Thr}(bc, cs, t)$ process synchronises on a set $\mathit{ThrChans}(t)$, which contains the events $\mathit{CEEswitchThread}.t.t2$ and $\mathit{CEEswitchThread}.t2.t$ for all thread identifiers $t2$. This ensures thread switches can be handled since the two threads involved in the switch (the thread switched from and the thread switched to) synchronise on the thread switch request. This model of the interpreter threads captures the fact that they are conceptually running in parallel, each with their own state, and we do not mandate a specific thread switch mechanism.

State

The state of each *Thr* process contains the stack for the thread, which consists of a series of stack frames, one for each method on the call stack. The contents of each stack frame are specified by the schema *StackFrame*. Its first component, *localVariables*, is a sequence of *Word* values representing the local variable array for the method. Its second component, *operandStack*, represents the data stack upon which each bytecode instruction operates. The third component, *storedPC*, is used for recording the program counter as a return address when another method is invoked. The fourth component, *frameClass*, is a copy of the *Class* information for the class of the stack frame's method, so that the constant pool for the class is available to the operations of *Thr*. The final component, *stackSize*, gives the maximum size of the *operandStack* for the thread.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StackFrame}} \\
\hline
\mathit{localVariables} : \operatorname{seq} \mathit{Word} \\
\mathit{operandStack} : \operatorname{seq} \mathit{Word} \\
\mathit{storedPC} : \mathit{ProgramAddress} \\
\mathit{frameClass} : \mathit{Class} \\
\mathit{stackSize} : \mathbb{N} \\
\hline
\#\,\mathit{operandStack} \leq \mathit{stackSize} \\
\hline
\end{array}
$$

The invariant of *StackFrame* just requires the *operandStack* to be no larger than *stackSize*.

The state of the *Thr* process is given by the schema *InterpreterState*, below. Its first component, *frameStack*, represents the stack itself, which is a sequence of *StackFrame* bindings, with one for each method entered. The second component, *pc*, is the program counter for the thread. Finally, the third component, *currentClass*, is a copy of the class information for the current method.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InterpreterState}} \\
\hline
\mathit{frameStack} : \operatorname{seq} \mathit{StackFrame} \\
\mathit{pc} : \mathit{ProgramAddress} \\
\mathit{currentClass} : \mathit{Class} \\
\hline
\mathit{frameStack} \neq \langle\rangle \Rightarrow \\
\quad \mathit{currentClass} = (\mathit{last}\,\mathit{frameStack}).\mathit{frameClass} \land \\
\quad \exists_1\, c : \operatorname{ran} \mathit{cs};\ m : \mathit{MethodID} \mid \\
\qquad m \in \operatorname{dom} c.\mathit{methodEntry} \land \\
\qquad \mathit{pc} \in c.\mathit{methodEntry}\, m \mathrel{..} c.\mathit{methodEnd}\, m \bullet \\
\qquad \mathit{currentClass} = c \\
\forall f : \operatorname{ran} \mathit{frameStack} \bullet \exists_1\, c : \operatorname{ran} \mathit{cs};\ m : \mathit{MethodID} \mid \\
\quad m \in \operatorname{dom} c.\mathit{methodEntry} \land \\
\quad f.\mathit{storedPC} \in c.\mathit{methodEntry}\, m \mathrel{..} c.\mathit{methodEnd}\, m \bullet \\
\quad f.\mathit{frameClass} = c \\
\hline
\end{array}
$$

The first conjunct of the invariant applies only if the *frameStack* is nonempty. It defines *currentClass* as the *frameClass* of the last *StackFrame* on the *frameStack*. It also requires that there is a unique class and method in *cs* containing *pc* in its *ProgramAddress* range, and that the class is *currentClass*. This ensures that the current class information can always be determined from the current *pc* value. The second conjunct of the invariant states a similar requirement for the *storedPC* value of each *StackFrame* in the *frameStack*, to ensure that the property holds for *currentClass* when a method returns.

The state is initialised as described in a schema *InterpreterInit*. The initialisation just sets the *frameStack* to empty. The other state components take arbitrary values and are initialised when the first *StackFrame* is created, since they are unused until then.

Behaviour

The main action of *Thr* is shown below. After the initialisation, it behaves as *MainThread* or *NotStarted*, depending on whether the thread represented by the *Thr* process is the *main* thread or not. *MainThread* and *NotStarted* make use of the same actions for executing bytecode instructions, but they occur in different orders. The control flow of the *Thr* process is shown in Figure 4.3.

$$
\bullet\ \big(\mathit{InterpreterInit}\big)\ ;\ \left(\begin{array}{l}
\big(\mathit{thread} = \mathit{main}\big)\ \&\ \mathit{MainThread} \\
\Box \\
\big(\mathit{thread} \neq \mathit{main}\big)\ \&\ \mathit{NotStarted}
\end{array}\right)
$$

The *MainThread* action is shown below. It begins by accepting a *StackID* from the *Launcher* on the *initMainThread* channel, ensuring that space has been allocated for the stack. It then offers a choice of executing a method on the *main* thread in response to a request from the *Launcher*, or switching to another thread. A request to start execution of a method is handled in the *StartInterpreter* action, which creates the *StackFrame* for the method. The process then polls the scheduler (discussed below, when we present the *Running* action) and behaves as the *Running* action, executing bytecode instructions until the method has finished. During method execution in *Running*, the thread may accept a request to switch to another thread, after which it behaves as *Blocked*, waiting for a request to switch back to the thread and continue execution

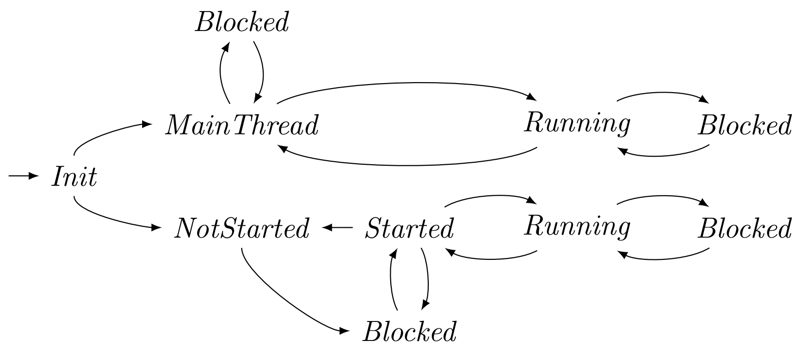


Figure 4.3: The overall control flow of *Thr*

in *Running*. When the execution of the method in *Running* has finished, the *MainThread* action recurses to offer the choice of method execution and thread switch again. If an instruction to switch to another thread from the scheduler is received on the *CEEswitchThread* channel, then it is only accepted if the thread switched from is the *thread* represented by the process. If it is accepted, then the process behaves as *Blocked*, after which it recurses back to offer the choice of behaviours again. Although the whole *Interpreter* model is the object of the compilation strategy, as we discuss in the next chapter, the action *Running* is the main focus.

$$
\begin{array}{l}
\mathit{MainThread} \mathrel{\widehat{=}} \mathit{initMainThread}?\mathit{stack} \longrightarrow \mu X \bullet \\
\qquad \left(\begin{array}{l}
\mathit{StartInterpreter} \;;\; \mathit{Poll} \;;\; \mathit{Running} \;;\; X \\
\Box \\
\mathit{CEEswitchThread}?\mathit{from}?\mathit{to} : (\mathit{from} = \mathit{thread}) \longrightarrow \mathit{Blocked} \;;\; X
\end{array}\right)
\end{array}
$$

The *StartInterpreter* action, used by *MainThread*, handles requests to execute a method from the *Launcher*. Its definition is shown below. It accepts communication on the *executeMethod* channel, requiring the *ThreadID* communicated to be the same as that of the current thread, *thread*, and storing the other values communicated as *classID*, *methodID* and *methodArgs*. A data operation *ResolveMethod* is then used to determine the appropriate class information for the method, since the method may actually be defined in a superclass of the provided *classID*. *ResolveMethod* follows the method resolution rules of the JVM specification, first checking if the class corresponding to *classID* defines the method, then checking if one of its superclasses defines the method, and finally looking for the method definition among its superinterfaces. The *Class* information resulting from this is stored in *class* and used to create a new *StackFrame* on the *frameStack* in *InterpreterNewStackFrame*.

$$
\begin{array}{l}
\mathit{StartInterpreter} \mathrel{\widehat{=}} \\
\quad \mathbf{var}\, \mathit{classID} : \mathit{ClassID};\ \mathit{methodID} : \mathit{MethodID};\ \mathit{methodArgs} : \operatorname{seq} \mathit{Word};\ \mathit{class} : \mathit{Class} \bullet \\
\quad \mathit{executeMethod}?t : (t = \mathit{thread})?c?m?a \longrightarrow \mathit{classID}, \mathit{methodID}, \mathit{methodArgs} := c, m, a; \\
\quad \big(\mathit{ResolveMethod}\big) \;;\; \big(\mathit{InterpreterNewStackFrame}\big)
\end{array}
$$

For threads other than *main*, the behaviour is described by the *NotStarted* action below. It accepts a request to start the *thread* represented by the process from the scheduler on the *CEEstartThread* channel. The identifier, *bsid*, of the thread's backing store is then passed to *ObjMan* via the *addThreadMemory* channel. The remaining information is stored in *classID*, *methodID* and *methodArgs*. The process then behaves as the *Blocked* action, waiting for an instruction to switch to that thread. After the thread is switched to, execution is passed to the *Launcher* via the *runThread* channel. The process then behaves as the *Started* action, which waits for a response from the *Launcher*.

$$\begin{array}{l}
\mathit{NotStarted} \mathrel{\widehat=} \mathbf{var}\, \mathit{methodID} : \mathit{MethodID};\ \mathit{methodArgs} : \operatorname{seq} \mathit{Word} \bullet \\
\quad \mathit{CEEstartThread}?\mathit{toStart}?\mathit{bsid}?\mathit{stack}?\mathit{cid}?\mathit{mid}?\mathit{args} : (\mathit{toStart} = \mathit{thread}) \\
\quad \longrightarrow \mathit{addThreadMemory}!\mathit{thread}!\mathit{bsid} \\
\quad \longrightarrow \mathit{methodID}, \mathit{methodArgs} := \mathit{mid}, \mathit{args}; \\
\quad \mathit{Blocked}\ ;\ \mathit{runThread}!\mathit{thread}!(\mathit{head}\ \mathit{methodArgs})!\mathit{methodID} \longrightarrow \mathit{Started}
\end{array}$$

The *Started* action, shown below, offers a choice of three behaviours. The first behaviour offered is accepting a request to execute a method as described by the *StartInterpreter* action, after which it behaves as the *Running* action, executing bytecode instructions. When *Running* terminates, a further choice is offered. The interpreter can continue to execute methods on the thread, signalled by the *continue* channel, after which the *Started* action recurses to offer the main choice again. The interpreter can also accept a request to end execution on the thread, signalled via the *endThread* channel, after which *Started* no longer offers the main choice and terminates the thread, as we describe below. The second action in the choice offered by *NotStarted* allows a switch away from *thread* to be accepted, after which the process behaves as *Blocked* before offering the choice again. The final action in the choice allows a request to terminate the thread to be accepted on the *endThread* channel. When the termination of the thread is requested, the thread's memory is removed using the *removeThreadMemory* channel and the scheduler is signalled on the *SendThread* channel, with a report of *Sokay* expected in response. This causes the scheduler to switch to a different thread, so a thread switch is accepted on the *CEEswitchThread* channel. After that, the process again behaves as *NotStarted*, allowing the thread to be restarted.

$$\begin{array}{l}
\mathit{Started} \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
\mathit{StartInterpreter}\ ;\ \mathit{Poll}\ ;\ \mathit{Running}\ ;\ \left(\begin{array}{l}
\mathit{continue}?t : (t = \mathit{thread}) \longrightarrow \mathit{Started} \\
\Box \\
\mathit{endThread}?t : (t = \mathit{thread}) \longrightarrow \mathbf{Skip}
\end{array} \right) \\
\Box \\
\mathit{CEEswitchThread}?\mathit{from}?\mathit{to} : (\mathit{from} = \mathit{thread}) \longrightarrow \mathit{Blocked}\ ;\ \mathit{Started} \\
\Box \\
\mathit{endThread}?t : (t = \mathit{thread}) \longrightarrow \mathbf{Skip}
\end{array} \right); \\
\quad \mathit{removeThreadMemory}!\mathit{thread} \longrightarrow \mathit{SendThread} \longrightarrow \mathit{Sreport}?r : (r = \mathit{Sokay}) \\
\quad \longrightarrow \mathit{CEEswitchThread}?\mathit{from}?\mathit{to} : (\mathit{from} = \mathit{thread}) \longrightarrow \mathit{NotStarted}
\end{array}$$

The *Blocked* and *Running* actions define the behaviour of threads after they have been started. The *Blocked* action simply waits for a signal on the *CEEswitchThread* channel to switch to *thread*, after which it terminates to allow execution to continue.

$$\mathit{Blocked} \mathrel{\widehat=} \mathit{CEEswitchThread}?\mathit{from}?\mathit{to} : (\mathit{to} = \mathit{thread}) \longrightarrow \mathbf{Skip}$$

The *Running* action, shown below, executes the bytecode instructions of a program. It has the form of a loop that repeatedly executes until *frameStack* is empty. Within the loop, it handles the bytecode instruction at the current *pc* value in *HandleInstruction* and then it polls for thread switches in *Poll*.

$$
\begin{array}{l}
\mathit{Running} \mathrel{\widehat=} \\
\quad \mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad \ \,[\!]\ \mathit{frameStack} \neq \varnothing \longrightarrow \mathit{HandleInstruction}\ ;\ \mathit{Poll}\ ;\ \mathit{Running} \\
\quad \mathbf{fi}
\end{array}
$$

Poll permits thread switches inbetween bytecode instructions. Implementations that allow thread switches at other points are valid if they retain the same sequence of externally visible events, meaning only instructions involving communication with other parts of the model need be atomic. *Poll* simply offers communication from the scheduler on the *CEEswitchThread* and *CEEproceed* channels, switching to *Blocked* upon receiving a signal on *CEEswitchThread*, and terminating on receiving a signal on *CEEproceed*.

The *HandleInstruction* action, shown in part below, offers a choice of actions for handling the bytecode instructions. There is one action for each of the instructions, with the action's name formed from the bytecode mnemonic prefixed with *Handle* (e.g. *HandleAload* for the `aload` instruction).

$$
\begin{array}{l}
\mathit{HandleInstruction} \mathrel{\widehat=} \\
\quad \mathit{HandleAconst\_null} \mathbin{\Box} \mathit{HandleDup} \mathbin{\Box} \mathit{HandleAload} \mathbin{\Box} \mathit{HandleAstore} \mathbin{\Box} \mathit{HandleIadd} \mathbin{\Box} \cdots
\end{array}
$$

The *Handle* actions define the semantics for the instructions and, as such, are involved in the compilation strategy. Many of these actions for handling bytecode instructions have a similar form.

Bytecode Semantics

The simplest *Handle* actions consist of a guard requiring the *bc* value at the current *pc* to be a particular bytecode instruction, followed by a data operation specified by a Z schema updating *InterpreterState*. This is illustrated in the definition of *HandleAconst_null* below, which uses the *InterpreterAconst_null* schema.

$$\mathit{HandleAconst\_null} \mathrel{\widehat=} (\mathit{pc} \in \operatorname{dom} \mathit{bc} \land \mathit{bc}\,\mathit{pc} = \mathit{aconst\_null}) \mathbin{\&} \big(\mathit{InterpreterAconst\_null}\big)$$

The *Circus* actions and Z schemas for each bytecode instruction are listed in Table 4.3. We omit the definitions of the Z schemas in our description here. They can be found in Appendix B of the extended version of this thesis [13]. Their contents are in line with the state updates for the bytecode instructions presented in Table 4.2.

The *HandleDup*, *HandleIadd* and *HandleIneg* actions follow the simple form exemplified above. Some instructions have parameters that must be extracted so that they can be passed to the data operation for the instruction. This can be seen in the definition of the *HandleAload* action, shown below, in which the inverse of the *aload* constructor is used to extract its parameter into a *variableIndex* variable that is used by the *IntepreterAload* schema.

$$
\begin{array}{l}
\mathit{HandleAload} \mathrel{\widehat=} (\mathit{pc} \in \operatorname{dom} \mathit{bc} \land \mathit{bc}\,\mathit{pc} \in \operatorname{ran} \mathit{aload}) \mathbin{\&} \\
\quad \mathbf{var}\ \mathit{variableIndex} : \mathbb{N} \bullet \mathit{variableIndex} := (\mathit{aload}^{\sim})\,(\mathit{bc}\,\mathit{pc})\ ;\ \big(\mathit{InterpreterAload}\big)
\end{array}
$$

The *HandleAstore*, *HandleGoto*, *HandleIconst* and *HandleIf_icmple* actions all follow a similar form, extracting the parameter of the bytecode instruction into a separate variable.

Bytecode instruction	*Circus* action	Z schema
`aconst_null`	$HandleAconst\_null$	$InterpreterAconst\_null$
`aload`	$HandleAload$	$InterpreterAload$
`areturn`	$HandleAreturn$	$InterpreterAreturn$
`astore`	$HandleAstore$	$InterpreterAstore$
`dup`	$HandleDup$	$InterpreterDup$
`getfield`	$HandleGetfield$	$InterpreterPop$, $InterpreterPush$
`getstatic`	$HandleGetstatic$	$InterpreterPush$
`goto`	$HandleGoto$	$InterpreterGoto$
`iadd`	$HandleIadd$	$InterpreterIadd$
`iconst`	$HandleIconst$	$InterpreterPush$
`if_icmple`	$HandleIf\_icmple$	$InterpreterIf\_icmple$
`ineg`	$HandleIneg$	$InterpreterIneg$
`invokespecial`	$HandleInvokespecial$	$InterpreterStackFrameInvoke$, $InterpreterNewStackFrame$
`invokestatic`	$HandleInvokestatic$	$InterpreterStackFrameInvoke$, $InterpreterNewStackFrame$
`invokevirtual`	$HandleInvokevirtual$	$InterpreterStackFrameInvoke$, $InterpreterNewStackFrame$
`new`	$HandleNew$	$InterpreterPush$
`putfield`	$HandlePutfield$	$InterpreterPop2$
`putstatic`	$HandlePutstatic$	$InterpreterPop$
`return`	$HandleReturn$	$InterpreterReturn$

Table 4.3: The relationship between the bytecode instructions in our subset and the *Circus* actions and Z schemas defining them

The actions to handle the return instructions (`areturn` and `return`), besides calling the Z data operation to deal with these instructions, must perform some additional operations. Firstly, the lock on the `this` object of the method must be released when returning from a synchronized method. This is handled by an additional action, *CheckSynchronizedReturn*, called before the data operation. Secondly, a return instruction has the possibility of engaging in a communication to pass the return value to the *Launcher* when returning from a method that has been started by the *Launcher*. This is performed by a second additional action, *CheckLauncherReturn*, which is called after return from the data operation. It checks whether the extra communication is needed and carries it out, if this is the case. These can be seen in the definition of *HandleAreturn* below, where *CheckLauncherReturn* is passed the return value, *returnValue*, from the *InterpreterAreturn* operation.

$$\begin{array}{l} HandleAreturn \mathrel{\widehat=} \mathbf{var}\ returnValue : Word \bullet (pc \in \operatorname{dom} bc \land bc\ pc = areturn) \& \\ \quad CheckSynchronizedReturn \mathbin{;} \big(InterpreterAreturn\big); \\ \quad CheckLauncherReturn(returnValue) \end{array}$$

The form of the *HandleReturn* action is similar, but since *InterpreterReturn* does not output a return value, *returnValue* takes an arbitrary value.

Within the *CheckSynchronizedReturn* action, shown below, the identifier of the current method, *methodID*, is obtained from the *pc* using a data operation, *GetCurrentMethod*. The information in *currentClass* is then checked to determine if *methodID* denotes a method that is synchronized and not static. We do not apply synchronisation to static methods, matching the behaviour of icecap, as we discuss in more detail in Section 5.2. For a method that is synchronized and not static, we communicate with the launcher on the *releaseLock* channel, sending the `this` pointer of the current method. The launcher then communicates with the scheduler to release the lock. If the method is static or not synchronized, then no communication is required and the action terminates.

```
$CheckSynchronizedReturn \mathrel{\widehat=}$
    $\mathbf{var}\ methodID : MethodID \bullet \big(GetCurrentMethod\big);$
    $\mathbf{if}\ methodID \in currentClass.synchronizedMethods$
        $\land\ methodID \notin currentClass.staticMethods \longrightarrow$
        $releaseLock!((last\ frameStack).localVariables\ 1) \longrightarrow releaseLockRet \longrightarrow \mathbf{Skip}$
    $[\!]\ methodID \notin currentClass.synchronizedMethods$
        $\lor\ methodID \in currentClass.staticMethods \longrightarrow \mathbf{Skip}$
    $\mathbf{fi}$
```

Within the *CheckLauncherReturn* action, the definition of which is shown below, the *frameStack* is checked to determine whether the return value should be communicated to the *Launcher*. If *frameStack* is empty then the method being returned from has been initiated via a signal on the *executeMethod* channel, and so the return value is communicated back to the *Launcher* via *executeMethodRet*. If the *frameStack* is not empty then nothing more needs to be done and the action terminates.

```
$CheckLauncherReturn \mathrel{\widehat=} \mathbf{val}\ returnValue : Word \bullet$
    $\mathbf{if}\ frameStack = \varnothing \longrightarrow executeMethodRet!thread!returnValue \longrightarrow \mathbf{Skip}$
    $[\!]\ frameStack \neq \varnothing \longrightarrow \mathbf{Skip}$
    $\mathbf{fi}$
```

For the instructions that create objects and access their fields (`new`, `getfield`, `putfield`, `getstatic` and `putstatic`), communication with *ObjMan* is needed. This can be seen in the definition of *HandleGetfield*, shown below, where the object identifier, *oid*, is popped from the *operandStack* of the current *StackFrame* using the data operation *InterpreterPop*, and the class identifier, *cid*, and the field identifier, *fid* are extracted from the parameter to the bytecode instruction. Note that pc and pc' are hidden in *InterpreterPop*, so that it does not change the value of pc, which is updated by *InterpreterPush*. The object, field and class identifiers are passed to *ObjMan* via the *getField* channel. The field's value is returned via the *getFieldRet* channel and pushed onto the *operandStack* of the current *StackFrame* by the data operation *InterpreterPush*.

```
$HandleGetfield \mathrel{\widehat=} \big(pc \in \mathrm{dom}\ bc \land bc\ pc \in \mathrm{ran}\ getfield\big)\&$
    $\mathbf{if}(getfield^{\sim})\ (bc\ pc) \in fieldRefIndices\ currentClass \longrightarrow$
        $\mathbf{var}\ oid : ObjectID \bullet \big(InterpreterPop[oid!/value!] \setminus (pc, pc')\big);$
        $\mathbf{var}\ fid : FieldID \bullet fid := fieldOf\ currentClass\ ((getfield^{\sim})\ (bc\ pc));$
        $\mathbf{var}\ cid : ClassID \bullet cid := (classOf\ currentClass\ ((getfield^{\sim})\ (bc\ pc)));$
        $getField!oid!cid!fid \longrightarrow getFieldRet?value \longrightarrow \big(InterpreterPush\big)$
    $[\!]\ (getfield^{\sim})\ (bc\ pc) \notin fieldRefIndices\ currentClass \longrightarrow \mathbf{Chaos}$
    $\mathbf{fi}$
```

The *HandleNew*, *HandlePutfield*, *HandleGetstatic* and *HandlePutstatic* actions are similar.

Finally, method invocation instructions (`invokespecial`, `invokestatic` and `invokevirtual`), require special handling by the virtual machine. Since the different method invocation instructions differ only in how the class for the method is determined and whether a `this` object identifier is passed among the method's arguments, the invocation of the method after this has been determined is handled by a common *Invoke* action. This can be seen in the definition of the *HandleInvokespecial* action below. The class identifier, *cid*, is extracted from the instruction's parameter and checked in a data operation *CheckSuperclass*. *CheckSuperclass* replaces *cid* with the identifier of the direct superclass of *currentClass* if it is a proper superclass of *currentClass* and *mid* does not refer to an initialisation method, in line with the semantics specified for the JVM. The method identifier *mid* is also extracted from the instruction's parameter, and the data operation *InterpreterStackFrameInvoke* is used to store the return *pc* address and pop the arguments of the method into *poppedArgs*. The number of arguments popped, *argsToPop*?, is the *methodArguments* value for *mid*, plus one for the `this` identifier passed to the method. The *cid* and *mid* identifiers are then passed into *Invoke* along with *poppedArgs*.

```
$HandleInvokespecial \mathrel{\widehat=}$
    $(pc \in \operatorname{dom} bc \land bc\,pc \in \operatorname{ran} invokespecial) \&$
    $\mathbf{var}\ cid : ClassID;\ mid : MethodID;\ poppedArgs : \operatorname{seq} Word \bullet$
    $\mathbf{if}((invokespecial^{\sim})\,(bc\,pc)) \in methodRefIndices\ currentClass \longrightarrow$
        $cid := classOf\ currentClass\ ((invokespecial^{\sim})\,(bc\,pc))\ ;\ (CheckSuperclass);$
        $mid := methodOf\ currentClass\ ((invokespecial^{\sim})\,(bc\,pc));$
        $(\exists\, argsToPop? == methodArguments\ mid + 1 \bullet InterpreterStackFrameInvoke);$
        $Invoke(cid, mid, poppedArgs)$
    $[\!]\ ((invokespecial^{\sim})\,(bc\,pc)) \notin methodRefIndices\ currentClass \longrightarrow \mathbf{Chaos}$
    $\mathbf{fi}$
```

The *HandleInvokestatic* and *HandleInvokevirtual* actions are similar, except that neither includes *CheckSuperclass*, *HandleInvokeStatic* does not include a `this` argument in *argsToPop*? , and *HandleInvokevirtual* obtains the class identifier from the type of the `this` object using the *getClassIDOf* channel rather than from the instruction's parameter.

The *Invoke* action, shown below, has the form of an external choice over actions for each of the special methods supported by the SCJVM, plus an *InvokeOther* action for handling non-special methods implemented in bytecode. The name of the action for each special method is formed from the name of the special method prefixed with *Invoke* (e.g. *InvokeResumeThread* for the `resumeThread()` method). The parameters passed to *Invoke* are passed on to each of the actions in the external choice.

$$
\begin{array}{l}
\mathit{Invoke} \mathrel{\widehat=} \mathbf{val}\, \mathit{classID} : \mathit{ClassID};\ \mathbf{val}\, \mathit{method} : \mathit{MethodID};\ \mathbf{val}\, \mathit{args} : \mathrm{seq}\, \mathit{Word} \bullet \\
\quad \mathit{InvokeSetPriorityCeiling}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeRegister}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeReleaseAperiodic}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeEnterPrivateMemory}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeExecuteInAreaOf}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeExecuteInOuterArea}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeExitMemory}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeInitAperiodicEventHandler}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeInitPeriodicEventHandler}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeInitOneShotEventHandlerAbs}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeInitOneShotEventHandlerRel}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeWrite}(\mathit{classID}, \mathit{method}, \mathit{args}) \\
\quad \Box\ \mathit{InvokeRead}(\mathit{classID}, \mathit{method}) \\
\quad \Box\ \mathit{InvokeOther}(\mathit{classID}, \mathit{method}, \mathit{args})
\end{array}
$$

Within the special-method actions, there is a guard ensuring the action is taken when the class and method identifiers are those for the method. The method is then handled by communication on the appropriate channels. This is illustrated by the definition of the *InvokeResumeThread* action, shown below. The class identifier parameter, *classID*, is required to refer to a subclass of some class *resumeThreadClass*, while the method identifier, *method*, must be *resumeThreadID*. The class and method identifiers used in the special method actions are a mixture of identifiers from the SCJ API and implementation-defined identifiers provided to expose SCJVM services to bytecode programs. The argument to the method, stored as the first element of the *methodArgs* parameter, is converted to a *ThreadID* and passed to the *Launcher* via the *resumeThread* channel. A return signal is then awaited on the *resumeThreadRet* channel before continuing.

$$
\begin{array}{l}
\mathit{InvokeResumeThread} \mathrel{\widehat=} \\
\quad \mathbf{val}\, \mathit{classID} : \mathit{ClassID};\ \mathbf{val}\, \mathit{method} : \mathit{MethodID};\ \mathbf{val}\, \mathit{methodArgs} : \mathrm{seq}\, \mathit{Word} \bullet \\
\quad \big((\mathit{classID}, \mathit{resumeThreadClass}) \in \mathit{subclassRel}\ \mathit{cs} \land \mathit{method} = \mathit{resumeThreadID}\big)\ \& \\
\quad \mathit{resumeThread}!(\mathit{WordToThreadID}\ (\mathit{methodArgs}\ 1)) \longrightarrow \mathit{resumeThreadRet} \longrightarrow \mathbf{Skip}
\end{array}
$$

In addition to the special methods handled in the *Launcher*, we also supply `read()` and `write()` methods for reading from and writing to some standard input and output devices. These methods are handled using the *input* and *output* channels that communicate the values from and to the environment of the SCJVM. This is shown in the definition of the *InvokeRead* action below, which accepts the input on the *input* channel and pushes it onto the stack as the return value for the method.

$$
\begin{array}{l}
\mathit{InvokeRead} \mathrel{\widehat=} \\
\quad \mathbf{val}\, \mathit{classID} : \mathit{ClassID};\ \mathbf{val}\, \mathit{method} : \mathit{MethodID} : \mathrm{seq}\, \mathit{Word} \bullet \\
\quad \big((\mathit{classID}, \mathit{readClass}) \in \mathit{subclassRel}\ \mathit{cs} \land \mathit{method} = \mathit{readID}\big)\ \& \\
\quad \mathit{input}?\mathit{value} \longrightarrow \big(\mathit{InterpreterPush} \setminus (\mathit{pc}, \mathit{pc}')\big)
\end{array}
$$

The *InvokeWrite* action is similar, writing the method argument to the *output* channel.

The *InvokeOther* action, shown below, describes the handling of non-special methods. It begins with a guard that is the conjunction of the negation of the guards for the invocation actions for the special methods. It starts execution of the method in the interpreter by first finding its *Class* information with *ResolveMethod*. We take the lock of the object pointed to by the first argument in *methodArguments* if the invoked method is synchronized. This is handled by an action *CheckSynchronizedInvoke*, which is similar to *CheckSynchronizedReturn*, but takes the class information, method identifier and method arguments as inputs, and performs its communication on the *takeLock* channel. A new *StackFrame* is then created with *IntepreterNewStackFrame*.

$$
\begin{array}{l}
\mathit{InvokeOther} \mathrel{\widehat=} \\
\quad \mathbf{val}\, \mathit{classID} : \mathit{ClassID};\ \mathbf{val}\, \mathit{methodID} : \mathit{MethodID};\ \mathbf{val}\, \mathit{methodArgs} : \mathrm{seq}\, \mathit{Word} \bullet \\
\quad (((\mathit{classID}, \mathit{setPriorityCeilingClass}) \notin \mathit{subclassRel}\ \mathit{cs} \\
\qquad\qquad \lor \mathit{methodID} \neq \mathit{setPriorityCeilingID}) \\
\qquad \land ((\mathit{classID}, \mathit{managedSchedulableClass}) \notin \mathit{subclassRel}\ \mathit{cs} \\
\qquad\quad \lor \mathit{methodID} \neq \mathit{registerID}) \\
\qquad \land ((\mathit{classID}, \mathit{aperiodicEventHandlerClass}) \notin \mathit{subclassRel}\ \mathit{cs} \\
\qquad\quad \lor \mathit{methodID} = \mathit{releaseAperiodicID}) \\
\qquad \land ((\mathit{classID}, \mathit{managedMemoryClass}) \notin \mathit{subclassRel}\ \mathit{cs} \\
\qquad\quad \lor \mathit{methodID} \notin \{\mathit{enterPrivateMemoryHelperID}, \mathit{executeInAreaOfHelperID}, \\
\qquad\qquad \mathit{executeInOuterAreaHelperID}, \mathit{exitMemoryID}\}) \\
\qquad \land ((\mathit{classID}, \mathit{aperiodicEventHandlerClass}) \notin \mathit{subclassRel}\ \mathit{cs} \\
\qquad\quad \lor \mathit{methodID} \neq \mathit{initAPEHID}) \\
\qquad \land ((\mathit{classID}, \mathit{periodicEventHandlerClass}) \notin \mathit{subclassRel}\ \mathit{cs} \\
\qquad\quad \lor \mathit{methodID} \neq \mathit{initPEHID}) \\
\qquad \land ((\mathit{classID}, \mathit{oneShotEventHandlerClass}) \notin \mathit{subclassRel}\ \mathit{cs} \lor \\
\qquad\quad \mathit{methodID} \neq \mathit{initOSEHAbsID}) \\
\qquad \land ((\mathit{classID}, \mathit{oneShotEventHandlerClass}) \notin \mathit{subclassRel}\ \mathit{cs} \lor \\
\qquad\quad \mathit{methodID} \neq \mathit{initOSEHRelID}) \\
\qquad \land ((\mathit{classID}, \mathit{readClass}) \notin \mathit{subclassRel}\ \mathit{cs} \lor \mathit{methodID} \neq \mathit{readID}) \\
\qquad \land ((\mathit{classID}, \mathit{writeClass}) \notin \mathit{subclassRel}\ \mathit{cs} \lor \mathit{methodID} \neq \mathit{writeID})) \mathbin{\&} \\
\quad \mathbf{var}\, \mathit{class} : \mathit{Class} \bullet \big(\mathit{ResolveMethod}[\mathit{cs}/\mathit{cs}?]\big); \\
\quad \mathit{CheckSynchronizedInvoke}(\mathit{class}, \mathit{methodID}, \mathit{methodArgs}); \\
\quad \big(\mathit{InterpreterNewStackFrame}\big)
\end{array}
$$

This concludes our description of the handling of bytecode instructions, and of our description of the CEE before the application of the compilation strategy. In the next section we describe the model of the C code that is used for the output of the compilation strategy.

4.4 C Code Model

As mentioned previously, the CEE after compilation to C has a similar structure to the CEE before compilation, but the object manager is replaced with a struct manager and the interpreter is replaced with the C program. The struct manager is represented by a process $\mathit{StructMan}_{cs}$, and the C program by a process $\mathit{CProg}_{bc,cs}$. These are placed in parallel composition with the *Launcher* process described in Section 4.2 to form a $\mathit{CCEE}_{bc,cs}$ process representing the CEE for a C program, as shown below.

$$\mathit{CCEE}_{bc,cs}(\mathit{sid}, \mathit{initOrder}) \mathrel{\widehat=} \mathit{StructMan}_{cs} \parallel \mathit{CProg}_{bc,cs} \parallel \mathit{Launcher}(\mathit{sid}, \mathit{initOrder})$$

The subscripts here indicate that the processes depend on the *bc* and *cs* constants used as inputs to the compilation strategy. However, *bc* and *cs* are not parameters of the processes.

The *instCS* parameter is also removed, but it is related to *bc* and so is not included as a separate subscript. We note that the *sid* and *initOrder* parameters to *Launcher* remain, since *Launcher* is not transformed during the compilation strategy.

The channels used for communication between these processes are the same as those in Table 4.1. We describe the $CProg_{bc,cs}$ process in Section 4.4.1. After that, in Section 4.4.2, $StructMan_{cs}$ is described.

4.4.1 Shallow Embedding of C in *Circus*

The C code output by our compilation strategy is represented by a *Circus* process $CProg_{bc,cs}$, which is determined by the bytecode instructions, *bc*, and the class information, *cs*. This process has a similar structure to that of *Interpreter*: a parallel composition of $CThr_{bc,cs}(t)$ processes representing C threads, one for each thread identifier *t* except the *idle* thread, as shown in the definition of $CProg_{bc,cs}$ below.

$$\textbf{process } CProg_{bc,cs} \mathrel{\widehat=} \|\, t : ThreadID \setminus \{idle\} \, [\![\, ThrChans(t)]\!] \bullet CThr_{bc,cs}(t)$$

$CThr_{bc,cs}$ has a similar structure to the *Thr* process in Section 4.3.4. However, the *pc* and *frameStack* components are eliminated from the state during compilation. The state of $CThr_{bc,cs}$ is thus empty.

The *Running* action and creation of stack frames (in *MainThread* and *Started*) are replaced with an *ExecuteMethod* action that executes the C function corresponding to a given method identifier. The main action of $CThr_{bc,cs}$ thus has the same structure as that of *Interpreter*, with a choice of *MainThread* for the *main* thread and *NotStarted* for non-*main* threads (see Figure 4.3). However, *MainThread* is now as shown below. This is similar to the definition of *MainThread* in *Thr*, but the information received from the *executeMethod* channel is passed into the *ExecuteMethod* action to select the correct C function to execute. After method execution has finished, the return value, *retVal*, is obtained from *ExecuteMethod* and communicated on the *executeMethodRet* channel.

$$
\begin{array}{l}
MainThread \mathrel{\widehat=} initMainThread?stack \longrightarrow \mu X \bullet \\
\quad \left(\begin{array}{l}
\textbf{var } retVal : Word \bullet executeMethod?t : (t = thread)?cid?mid?args \longrightarrow \\
\qquad ExecuteMethod(cid, mid, args, retVal); \\
\qquad executeMethodRet!retVal \longrightarrow X \\
\Box \\
CEEswitchThread?from?to : (from = thread) \longrightarrow Blocked \,;\; X
\end{array} \right)
\end{array}
$$

The sequential composition of *StartInterpreter* and *Running* in *Started* is replaced with a call to the action *ExecuteMethod* in the same way as for the same sequential composition in *MainThread* shown above.

The *ExecuteMethod* action has the form shown below. It takes as parameters the class identifier, *cid*, method identifier, *mid*, and arguments list, *args*, for the method to be executed. It then chooses the appropriate action corresponding to the supplied *cid* and *mid*, and passes the appropriate number of arguments from *args* to the action. The return value of each of the actions, if they return one, is captured in *retVal* to be returned to *MainThread* or *NotStarted*. A function with both a return value and arguments has its value parameters (representing the arguments) followed by the result parameter (representing the return value).

$$
\begin{array}{l}
ExecuteMethod \mathrel{\widehat=} \\
\quad \mathbf{val}\, cid : ClassID;\ \mathbf{val}\, mid : MethodID;\ \mathbf{val}\, args : \mathrm{seq}\, Word;\ \mathbf{res}\, retVal : Word \bullet \\
\quad \mathbf{if}\,(cid, mid) = (<classID_1>, <methodID_1>) \longrightarrow \\
\qquad <classID_1>\_<methodID_1>(args\,1, \ldots, args\,(methodArgs\,<methodID_1>), retVal) \\
\qquad \vdots \\
\quad [\!]\ (cid, mid) = (<classID_n>, <methodID_{m_n}>) \longrightarrow \\
\qquad <classID_n>\_<methodID_{m_n}>(args\,1, \ldots, args\,(methodArgs\,<methodID_{m_n}>), retVal) \\
\quad \mathbf{fi}
\end{array}
$$

The actions used by *ExecuteMethod* represent C functions embedding the behaviour of the compiled methods. The name of each action is made up of the class and method identifier for the method, separated by an underscore. Within the action, the constructs of C are represented by constructs of *Circus*. The representation of these constructs is summarised in Table 4.4. The *Circus* code resulting from our compilation strategy can be converted to C by matching the patterns shown in Table 4.4 over the *Circus* syntax tree, and indeed we do so in a prototype implementation of our compilation strategy, described in Section 6.3.

The constructs we allow are conditionals, while loops, assignment statements, and function calls. These are comparable with those allowed in MISRA-C [82] and present in the code generated by icecap. Conditionals in C correspond to *Circus* alternation blocks, similar to those in Dijkstra's guarded command language [33]. We handle loops using recursion, with alternation used to handle loop conditions.

As each function in the C code is a *Circus* action, function calls are represented as references to those actions. Function arguments in C are passed by value, although those values may be pointers to other values. Accordingly, since our SCJVM model represents pointers explicitly (via the object or struct manager), we represent function arguments using value parameters of the *Circus* action.

If a function has a return value, it is represented with a result parameter of the *Circus* action, usually named *retVal*, with an assignment to that parameter at the end of the action representing return statements. In the C code resulting from our strategy, we represent these result parameters using pointers passed into the function, rather than C return values. We follow this representation rather than that of of icecap, since icecap passes values using a stack represented by a pointer passed to each function. That approach is used in icecap to provide for interaction between interpreted and compiled code, which we do not require in our code. Also, while we do not consider them in our compilation strategy, it may be noted that this approach scales well to `long` values, which occupy two variables. We follow guidelines for safety-critical uses of C variants, such as MISRA-C [82], and use a single return statement at the end of a function.

Local variables are represented using *Circus* variable blocks. These are placed after the parameter declarations. While *Circus* variable blocks could also be used to represent variables declared in the middle of functions, that is not necessary for our work. Restricting ourselves to variables at the start of functions ensures the code our strategy generates is compatible with older versions of C.

The types of parameters and variables in our *Circus* model is *Word*, representing the type of 32-bit JVM words. The corresponding type we use in C is `int32_t`, the type of 32-bit signed

Construct	C code	*Circus* equivalent
Function definition	`void foo() {...}`	$Foo \mathrel{\widehat=} \cdots$
Function definition with argument	`void bar(int32_t x) {...}`	$Bar \mathrel{\widehat=} \mathbf{val}\, x : Word \bullet \cdots$
Function definition with return value	`void baz(int32_t * retVal) {...}`	$Baz \mathrel{\widehat=} \mathbf{res}\, retVal : Word \bullet \cdots$

Function definition with parameter and return value	`void quux(int32_t x,`
	`    int32_t * retVal) {...}`
		$Quux \mathrel{\widehat=} \mathbf{val}\, x : Word;$
		$\mathbf{res}\, retVal : Word \bullet \cdots$

Function call	`foo();`	Foo
Function call with argument	`bar(x);`	$Bar(x)$
Function call with return value	`baz(& x);`	$Baz(x)$
Function call with argument and return value	`quux(x, & y);`	$Quux(x, y)$
Return statement	`return;`	$\mathbf{Skip}$

Return statement with value	`*retVal = x;`
	`return;`
		$retVal := x$

Assignment	`x = e;`	$x := e$
Variable declaration	`int32_t x;`	$\mathbf{var}\, x : word \bullet$
Variable declaration and initialisation	`int32_t x = e;`	$\mathbf{var}\, x : Word \bullet x := e$
If statement	`if (b) {...}`	$\mathbf{if}\ b \longrightarrow \cdots$ $[\!]\ \neg\, b \longrightarrow \mathbf{Skip}$ $\mathbf{fi}$
If-else statement	`if (b) {...} else {...}`	$\mathbf{if}\ b \longrightarrow \cdots$ $[\!]\ \neg\, b \longrightarrow \cdots$ $\mathbf{fi}$
Infinite loop	`while (1) {...}`	$\mu X \bullet \cdots ;\ X$
While loop	`while (b) {...}`	$\mu X \bullet$ $\mathbf{if}\ b \longrightarrow \cdots ;\ X$ $[\!]\ \neg\, b \longrightarrow \mathbf{Skip}$ $\mathbf{fi}$
Do-while loop	`do {...} while (b);`	$\mu X \bullet \cdots ;$ $\mathbf{if}\ b \longrightarrow X$ $[\!]\ \neg\, b \longrightarrow \mathbf{Skip}$ $\mathbf{fi}$
Field read	`y = ((C *) (uintptr_t)  x)->f;`	$getField!x!C!f \longrightarrow$ $getFieldRet?value \longrightarrow$ $y := value$
Field write	`((C *) (uintptr_t) x)->f = y;`	$putField!x!C!f!y \longrightarrow \mathbf{Skip}$

Table 4.4: The *Circus* representations of C constructs in our shallow embedding

integers provided by C99. This matches the behaviour of icecap, which uses its own `int32` type defined to be the same as `int32_t`. We note that we do not need to address general issues of mapping between C types and *Circus* types here, since all the variables and stack slots in the JVM are of the same fixed-width type.

Finally, we model accesses to structs representing objects using communications with the struct manager. These use the same channels as in the interpreter model, but the struct manager model, described in the next section, is such that the communications have the effect of performing reads and updates of object structs, with appropriate C casts and dereferences of pointers to such structs. The object, field and class identifiers required for these struct accesses are included in the channel communications so, with the semantics conferred by the struct manager, they can be translated to the corresponding C code by a simple lexical transformation, as shown in Table 4.4.

Note that our C code for accesses to structs involves a cast from `int32_t` to a pointer type. In order to ensure this cast is performed correctly on systems where pointers are not 32-bit, we also perform a cast to `uintptr_t`, the unsigned integer type from C99 with the same width as a pointer. This matches icecap, where a cast to a `pointer` type, defined in the same way as `uintptr_t` is performed as part of struct accesses. Accesses to static fields are performed similarly, using the *getStatic* and *putStatic* channels to represent accesses to a global static fields struct. The struct types and the functions for manipulating them are described in the next section, where we discuss the struct manager, $StructMan_{cs}$.

4.4.2 Struct Manager

$StructMan_{cs}$ manages objects represented by C structs that incorporate the class information from cs. $StructMan_{cs}$ has Z schemas representing struct types for objects of each class. For each class identifier $<classID_1>, \ldots, <classID_n>$, we define a schema $<classID_k>Obj$ for $k \in \{1, \ldots, n\}$, representing the objects of that class. They begin with a *classID* component containing the class identifier of the object, so that polymorphic method calls can be made by choice over the object's class. There is then a component for each of the fields $<fieldID_{k,1}>, \ldots, <fieldID_{k,m_k}>$, each of type *Word*.

$$\begin{array}{|l}
\hline
<classID_k>Obj \\
\hline
classID : ClassID \\
<fieldID_{k,1}> : Word \\
\quad \vdots \\
<fieldID_{k,m_k}> : Word \\
\hline
\end{array}$$

The schema types for each type of object are combined into a single free type *ObjectStruct*. The constructor for each $<classID_k>$ is called $<classID_k>Con$, with a single parameter of type $<classID_k>Obj$.

$$\begin{array}{l}
ObjectStruct ::= \\
\quad <classID_1>Con\langle\langle <classID_1>Obj \rangle\rangle \mid \ldots \mid <classID_n>Con\langle\langle <classID_n>Obj \rangle\rangle
\end{array}$$

For each object type, we define a natural number constant $sizeof<classID_k>Obj$ that represents the result of applying C's `sizeof` operator to the struct represented by the corresponding

$<\mathit{classID}_k>\mathit{Obj}$ type. We also define a function *classIDOf* for obtaining the value of the common *classID* field from an *ObjectStruct* value. Additionally, we define a $\mathit{cast}<\mathit{classID}_k>$ function for each $<\mathit{classID}_k>$, which maps an *ObjectStruct* value to a $<\mathit{classID}_k>\mathit{Obj}$ value. This works not only for values in the range of the $<\mathit{classID}_k>\mathit{Con}$ constructor, but also for any class that is a subclass of $<\mathit{classID}_k>$, with the common fields copied across. Thus, $\mathit{cast}<\mathit{classID}_k>$ represents casting of C structs, where a struct can be truncated by casting to a struct whose fields are a prefix of it. Finally, we define a function $\mathit{update}<\mathit{classID}_k>\_<\mathit{fieldID}>_i$ for each class $<\mathit{classID}_k>$ and field identifier $<\mathit{fieldID}_i>$, which takes an *ObjectStruct* and updates the field with a given value. This is a combined cast and update.

The static fields $<\mathit{staticFieldID}_{i,j}>$ from each class $<\mathit{classID}_i>$ are collected together in a schema *StaticFields*, as shown below.

$$\begin{array}{|l}
\hline
\mathit{StaticFields} \\
<\mathit{classID}_1>\_<\mathit{staticFieldID}_{1,1}> : \mathit{Word} \\
\quad\vdots \\
<\mathit{classID}_1>\_<\mathit{staticFieldID}_{1,\ell_1}> : \mathit{Word} \\
\quad\vdots \\
<\mathit{classID}_n>\_<\mathit{staticFieldID}_{n,1}> : \mathit{Word} \\
\quad\vdots \\
<\mathit{classID}_n>\_<\mathit{staticFieldID}_{n,\ell_n}> : \mathit{Word} \\
\hline
\end{array}$$

We define a constant *sizeofStaticFields* giving the space needed for the struct represented by *StaticFields*. Functions $\mathit{updateStatic}<\mathit{classID}_i>\_<\mathit{staticField}_{i,j}>$ are also defined for each class and static field to perform updates of specific fields in the *StaticFields* struct.

The state of $\mathit{StructMan}_{cs}$ is given by the schema *StructManState*. It is similar to *ObjManState* defined in Section 4.3.3, but the *objects* map relates object identifiers to *ObjectStruct* values, and *staticClassFields* is of the *StaticFields* type.

The structure of the $\mathit{StructMan}_{cs}$ process is much the same as for the *ObjMan* process, with the state initialised in a similar way. However, the initialisation of *staticClassFields* is done in terms of the *StaticFields* type, although the fields are still set to *null*, so *Init* is refined to reflect that.

Also, the actions *GetField*, *PutField*, *GetStatic* and *PutStatic* are refined to operate on the struct types described in this section. *GetField* simply applies the $\mathit{cast}<\mathit{cid}>$ function for the *classID* value *cid* provided on the *getField* channel to the object corresponding to the *ObjectID* provided on the *getField* channel, and returns the requested field of the resultant struct on *getFieldRet*. Similarly, *PutField* updates the specified object using $\mathit{update}<\mathit{cid}>\_<\mathit{fid}>$ for the *ClassID cid* and *FieldID fid* provided on the *putField* channel. The *GetStatic* and *PutStatic* actions access and update the specified static field in *staticClassFields*. We omit the definitions of these actions here; their definitions can be found in Appendix B of the extended version of this thesis [13], where we show the general form of the $\mathit{StructMan}_{cs}$ process.

The *NewObject* action is different in $\mathit{StructMan}_{cs}$ to that in *ObjMan*. It uses the same channels (*newObject* and *newObjectRet*), but creates an *ObjectStruct* value for the provided class. It has the form shown below. The *thread* and *classID* identifiers are received through *newObject* like in *ObjMan*. A choice is then made over the *classID*, matching it against each identifier supported by $\mathit{StructMan}_{cs}$. If *classID* matches an identifier $<\mathit{classID}_k>$, then space for the object is allocated via communication with the memory manager, as in *AllocateStaticFields*, and finally the object is stored in *objects* and initialised. The allocation is performed in a separate action, *AllocateObject*, as it is similar for each class. The size of the object is given by the $sizeof{<}classID_k{>}$ identifier for ${<}classID_k{>}$, and the returned object identifier is stored in *objectID*. The storing and initialisation of the object is defined by a schema action $StructMan{<}classID_k{>}ObjInit$, which sets all the object's fields to *null* and puts it in *objects*, stored within ${<}classID_k{>}Con$. Finally, *objectID* is returned via *newObjectRet*, as in *ObjMan*. The possibility of divergence if the memory manager reports an error is handled in *AllocateObject*.

```
$NewObject \mathrel{\widehat=} \mathbf{var}\ objectID : ObjectID \bullet$
    $newObject?thread?classID \longrightarrow$
    $\mathbf{if}\ classID = {<}classID_1{>} \longrightarrow$
        $AllocateObject(thread, sizeof{<}classID_1{>}, objectID);$
        $(StructMan{<}classID_1{>}ObjInit)$
     $[\!]\ classID = {<}classID_2{>} \longrightarrow$
        $AllocateObject(thread, sizeof{<}classID_2{>}, objectID);$
        $(StructMan{<}classID_2{>}ObjInit)$
        $\vdots$
     $[\!]\ classID = {<}classID_n{>} \longrightarrow$
        $AllocateObject(thread, sizeof{<}classID_n{>}, objectID);$
        $(StructMan{<}classID_n{>}ObjInit)$
    $\mathbf{fi}\ ;\ newObjectRet!objectID \longrightarrow \mathbf{Skip}$
```

Finally, *GetClassIDOf* is changed to extract the class identifier from an *ObjectStruct* value using the *classIDOf* function. This represents a C cast to a struct type representing the `Object` class, which is always valid since every class extends `Object`, and an access of the *class* field of that struct type.

This concludes our explanation of the model for the C code. In the next section we discuss how the models of the core execution environment before and after compilation can be validated.

4.5 Validation

It is important that our model provides an accurate representation of the semantics of an SCJVM. In creating this model we have carefully read the SCJ specification and the JVM specification, to extract the requirements for an SCJVM. During this process, we have had contact with the expert group developing the SCJ specification, who have been able to clarify several points about SCJ. This has resulted in several changes to the SCJ specification. This is the first piece of evidence that our model not only reflects the SCJ standard, but in some cases the standard has been changed to reflect our model. We list below the aspects of the SCJ standard that have been influenced by our work.

Though interrupts logically behave as small high-priority threads, it was not made clear in the SCJ specification what the current schedulable object should be during an interrupt. It has now been clarified that it is an error to request the current schedulable object while in an interrupt.

It was not clear what backing stores should be created during mission setup and what their sizes should be. This caused the SCJ expert group to review how the memory model from RTSJ, upon which SCJ is built, interacts with the SCJ mission model. The parameters to the classes that form part of SCJ's mission model have now been clarified in the SCJ specification to indicate the amount of backing store space each requires.

We also found that it was not clear in the SCJ specification that the instance of the class implementing `Safelet` requires initialisation, although the JVM specification states that every object must be initialised. This has now been clarified in the SCJ specification, and led to discussion about how command-line arguments are passed to an SCJ program. As a result, a `String[]` argument has been added to the `initializeApplication()` method of `Safelet`. We have not included this in our model as it did not appear in a publicly available version of the SCJ specification in time to be integrated into our model and none of our examples require command line parameters.

Some of our consultations with the SCJ expert group have provided clarification that we used to shape our model but did not result in changes to the SCJ specification. An example of such a clarification is concerning what happens when an alarm with a past time is set on the system real-time clock: it is an error although this is not made clear in the SCJ specification, since the response to it is implementation-defined. We have also discussed what should happen when a release of an event handler starts while a previous release is still running. It was established that a new release should start after the end of the existing release, but that comes from information in RTSJ and so did not result in changes to the SCJ specification. Since we have determined that this should be handled as part of the SCJ API implementation, it is not reflected in our model, which provides lower-level scheduling services that are used by the SCJ API. It is also not made clear whether the service of suspending a thread is available at SCJ level 1, but the implementation of an SCJ API class may make use of it though it is not made available to the application. We have also checked with the SCJ expert group to ensure the pattern of communication between the scheduler and CEE is in line with the requirements of SCJ.

In addition to the checking of our model against informal requirements, the model has been written using Community Z Tools (CZT) [71], which provides parsing and typechecking for *Circus* [70]. We have also performed some proofs on the Z parts of our model using Z/EVES [100]. These proofs are domain check proofs and precondition proofs. They ensure that an implementation of the model is possible and give further assurance that the model is sensible.

Finally, the compilation strategy presented in the next chapter provides further validation of our model, since we can check if the expected C code is produced by our strategy. Our compilation strategy consists of individual compilation rules, which are proved from laws whose correctness has been previously established, as explained in Section 6.2. We can thus have confidence that the semantics of the C code that results from the strategy has the same semantics as the bytecode input to our interpreter model. In addition, since we also produce a prototype implementation of this compilation strategy, described in Section 6.3, we can easily produce the output of the strategy and compare it to the corresponding code produced by icecap; indeed we do so for some examples in Section 6.4. Since the generated C code corresponds to the semantics of the bytecode in the interpreter model, this validates the semantics in our interpreter model.

Due to all these reasons, we can have confidence that our SCJVM model is correct. In the next section we conclude the chapter with some final discussion of additional points of interest concerning the model.

4.6 Final Considerations

In this chapter we have presented our model of the CEE of an SCJVM and specified the subset of Java bytecode covered in our model. Our bytecode subset focusses on method invocation and the manipulation of objects, since those are core concepts of Java. We have omitted instructions for exception handling, since that would complicate the model while adding little power. Our subset is sufficiently small to permit reasoning, but large enough to express a variety of SCJ programs.

Our CEE model is divided in three components, with a *Circus* process representing each component. The first component is the object manager, *ObjMan*, which manages objects and the entering of backing stores, since the memory manager discussed in the previous chapter has no knowledge of the structure of objects. The second component of the CEE model is the *Interpreter*, which describes the semantics of each of the bytecode instructions in our subset and provides for executing methods. The third and final component is the *Launcher*, which manages the SCJ mission model and coordinates execution.

One interesting point about our model is the handling of special methods in the *Interpreter* and *Launcher*. This is necessary for several reasons: to allow methods running in the interpreter to access the SCJVM services defined in the previous chapter, to allow mission setup methods to interact with the launcher, and to permit entering of memory areas via interaction with the CEE object manager. The handling of special methods works by having the interpreter check upon invocation of a method whether it requires special handling. If it does, the invocation is passed to the launcher to be handled. The launcher then communicates with the SCJVM services and the object manager as required.

After the compilation strategy has been applied, these special method calls become communications with the *Launcher*, representing calls to C functions in the SCJVM infrastructure. A similar approach could be used to handle native method calls, though we view that as future work since it is not a central part of the considerations for an SCJVM. Native methods can be represented via a shallow embedding in *Circus*, in the same way as the output of the compilation, but before compilation special handling can be carried out via calls to them in the interpreter.

The real-time requirements on SCJ scheduling also impose predictability, so that the bytecode instructions processed by the interpreter must appear to be atomic. This is specified in our model by only permitting thread switches inbetween bytecode instructions. This atomicity requirement is preserved throughout our compilation strategy, and the behaviour of polling for thread switches remains inbetween the C code corresponding to each bytecode instruction.

However, a correct implementation is required only to have the same sequence of externally visible events as our C code model. Many of the bytecode instructions only affect the state of the current thread, and so a thread switch in the middle of such an instruction would appear to an external observer the same as a thread switch just before or after them. The bytecode instructions which have effects visible outside the *Interpreter*, which are the `new` instruction, the field access instructions, and instructions that invoke the special methods mentioned above, interact with shared memory. Thread switches must not occur in the middle of these instructions to avoid leaving shared memory in an erroneous state. Only an implementation that ensures such operations are not interrupted, usually by employing synchronisation, is a correct implementation of our model. This is, of course, the case for icecap.

The main purpose of the model presented in this chapter is as a specification of the source and target languages for the compilation strategy presented in the next chapter. However, there are also other possible uses for it. For example, it can be used as a specification for an implementation of an interpreting SCJVM. Such an SCJVM could incorporate the compilation strategy to provide a choice between interpreted and complied code, as in the icecap HVM. Additionally, since error handling in our model is done via aborting execution, an identification of the conditions required for the model to be divergence-free produces requirements that can be used for bytecode verification.

Chapter 5

Compilation Strategy

In this chapter we describe our compilation strategy for refining SCJ bytecode to C code. We begin in Section 5.1 with an overview of our compilation strategy. Then, in Section 5.2 we describe the requirements on the source program for the compilation strategy to be applied. Afterwards, we describe each stage of the strategy in a separate section. The first stage, which we call *Elimination of Program Counter*, is described in Section 5.3. The second stage, called *Elimination of Frame Stack*, is described in Section 5.4. Finally, the third stage of the strategy, which is called *Data Refinement of Objects*, is described in Section 5.5. We then show how the stages fit together to show the compilation as a whole to be correct in Section 5.6, and conclude with some final considerations in Section 5.7.

5.1 Overview

Our compilation strategy refines the $CEE(bc, cs, instCS, sid, initOrder)$ process defined in Section 4.3 to obtain the $CCEE_{bc,cs}(sid, initOrder)$ process in Section 4.4. The overall theorem for the strategy, and, therefore, the main result presented in this chapter, is as follows.

Theorem 5.1.1 (Compilation Strategy). *Given bc, cs and sid, there are processes $StructMan_{cs}$ and $CProg_{bc,cs}$ such that,*

$$CEE(bc, cs, instCS, sid, initOrder) \sqsubseteq StructMan_{cs} \parallel CProg_{bc,cs} \parallel Launcher(sid, initOrder).$$

$StructMan_{cs}$ manages objects represented by C structs that incorporate the class information from cs, refining the process $ObjMan$, which handles abstract objects. $CProg_{bc,cs}$ refines the $Interpreter$, with the Thr processes refined into the $CThr_{bc,cs}$ processes described in Section 4.4.1. This means that the threads from SCJ are mapped onto threads in C, since we do not dictate a particular thread switch mechanism in either the source or target models.

The compilation strategy is split into three stages. Each stage has a theorem describing it, for which the strategy acts as a proof. The proof of Theorem 5.1.1, presented in Section 5.6, is obtained by an application of the theorems for each stage. Each stage of the compilation strategy handles a different part of the $Interpreter$ state: the pc, the $frameStack$, and objects. They operate over each of the Thr processes, managed by the SCJVM services.

The first stage, *Elimination of Program Counter*, introduces the control constructs of the C code. This removes the use of pc to determine the control flow of the program. The choice over *pc* values is replaced with a choice over method identifiers pointing to sequences of operations representing method bodies.

In the second stage, *Elimination of Frame Stack*, the information contained on the *frameStack*, which is the local variable array and operand stack for each method, is introduced in the C code. This is done by introducing variables and parameters to represent each method's local variables and operand stack slots. A data refinement is then used to transform each operation over the *frameStack* to operate on the new variables. The *frameStack* is then eliminated from the state.

In the final stage, *Data Refinement of Objects*, the class information from *cs* is used to create a representation of C structs. This means that *ObjMan*, which has a very abstract representation of objects, is transformed into *StructMan*. The operations on objects are then changed to access the structs for the objects in a more concrete way that represents the way struct fields are accessed in C code.

5.2 Assumptions about source bytecode

For our strategy to be successfully applied to bytecodes corresponding to an SCJ program, it must meet some basic requirements that ensure it is well-formed. Firstly, the program must pass JVM bytecode verification. This means it must be type-correct and that execution remains inside the array of bytecode instructions for each method. This can be checked before execution of the program and there has already been much work on formal verification of bytecode verifiers [29, 53, 58].

Secondly, since SCJ does not allow dynamic class loading, all required classes and methods must be present before execution of the program. This means that the *cs* map provided as input to the CEE must contain all the classes referenced by any other class in *cs*. All the bytecode instructions required for these classes must also be present in the *bc* map. Our CEE model diverges if any of these requirements is not met, so these requirements hold for any SCJ program that executes correctly in our SCJVM interpreter.

Thirdly, due to the nature of the applications that SCJ is aimed at, it is important that they have a structure that is readable and facilitates verification. MISRA-C includes such a restriction on structure and, since we are generating C code for a safety-critical application, we aim to produce code that is compatible with MISRA-C. This means that the SCJ bytecode program used as input to the strategy must also have a control structure compatible with the requirements of MISRA-C.

Precisely, we require the control flow graph of each method in the input program to have a structure based on Dijkstra's notion of program structure found in [32]. In our definition of a structured program, the control flow graph must be composed of the structures shown in Figure 5.1. The first structure (Figure 5.1a) is that of simple sequential composition, with an edge going from the root node to a single end node. The next three structures (Figure 5.1b–d) are conditional structures. Figure 5.1b shows an `if` statement with no `else` clause. Figure 5.1c shows an `if` statement with an `else` clause. Figure 5.1d shows a conditional in which both branches end with a (infinite) loop or a return so that there is nothing following the conditional; we refer to such conditionals as divergent conditionals since the branches do not come back together. The remaining three structures (Figure 5.1e–g) are all loops. Figure 5.1e shows a loop in which the loop condition is checked at the beginning (a `while` loop). Figure 5.1f shows

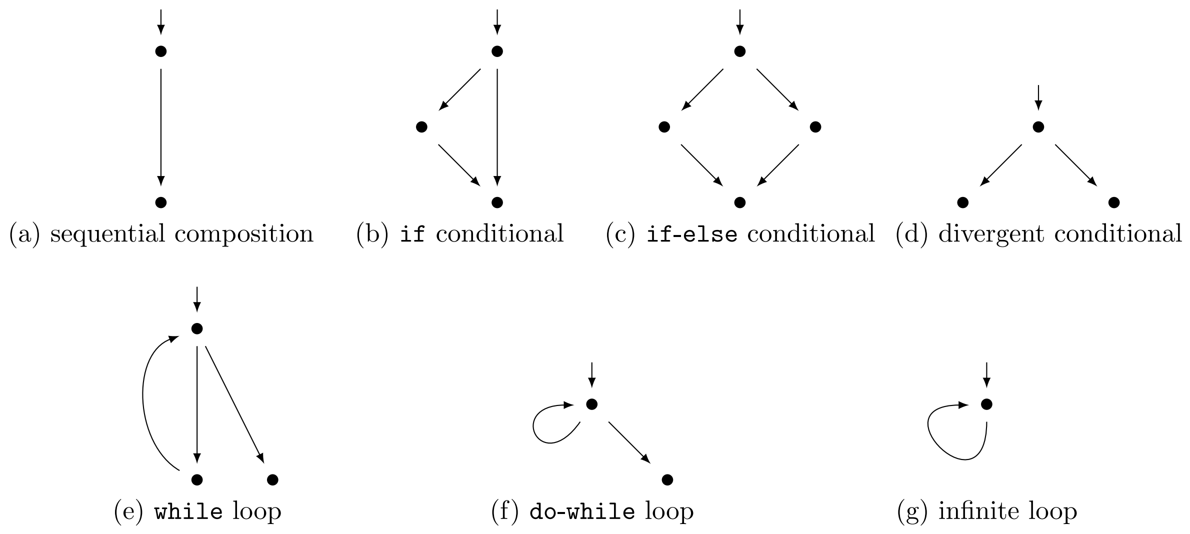


Figure 5.1: Control flow graphs of program structures

a loop in which the loop condition is checked at the end (a `do-while` loop). Figure 5.1g shows an infinite loop.

We provide below a formal definition of what it means for a control flow graph to be structured. This definition is based on that in [15], which provides an algorithm for recognising structured graphs. We first define a rooted directed graph below. The definition is standard, but we include it here to introduce the terminology for the subsequent definition.

Definition 5.2.1 (Rooted Directed Graph)**.** A *rooted directed graph*, G, is a triple (V, E, r), where

- V is a set of *nodes*,
- E is a set of ordered pairs of nodes in V, called *edges*, and
- r is a node in V, called the *root* of the graph.

The first component of an edge is its *source* and the second component is its *target*. We say that an edge goes from its source to its target. For every node $n \in V$, the pair (r, n) must be in the reflexive transitive closure of E, that is, there must be a path of edges from the root to any node in the graph. For a graph G, we refer to the set $T(G) = \{n \in V \mid \forall\, m \in V.\ (n, m) \notin E\}$ of nodes with no edges coming from them as the set of *end nodes* of the graph.

In diagrams we represent the nodes as points or as the names of the nodes, the edges as arrows, and the root node as a node with an arrow pointing to it that does not come from another node. Additionally, we refer to the source of an edge going to a given node as a *predecessor* of that node; similarly, the target of an edge from a given node is a *successor* of that node.

We now define what it means to replace a node in a graph by another graph. We use this concept to construct more complex structured graphs from those shown in Figure 5.1. Node replacement may occur in four different ways, depending on which node is being replaced in a graph. We illustrate the different cases of node replacement using the example graphs G and H shown in Figure 5.2. The G graph has the form of a conditional with two branches, and the

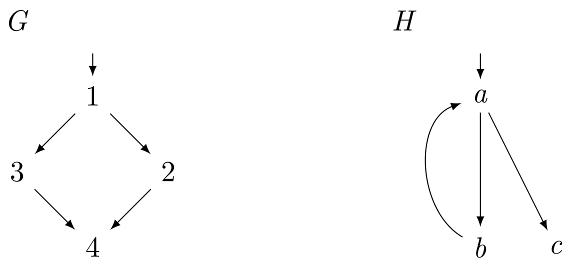


Figure 5.2: Example control flow graphs to illustrate node replacement

H graph has the form of a `while` loop. We label the nodes of the graphs separately for ease of reference.

The first case is that of placing a graph at the start of another graph, i.e. replacing the root node of a graph that does not have a loop to its root node. An example of this can be seen in Figure 5.3a, where the root node (node 1) of graph G is replaced with graph H. The unique end node of graph H, node c, takes the place of node 1. The other nodes of H are connected to it by the same edges as in H.

The second case is that of replacing one of the end nodes of a graph. This is shown in Figure 5.3b, where node 4 of graph G is replaced with graph H. Node a, the root node of graph H, takes the place of node 4. As in the previous case, the remaining nodes of H are included, connected to a by the same edges as in H.

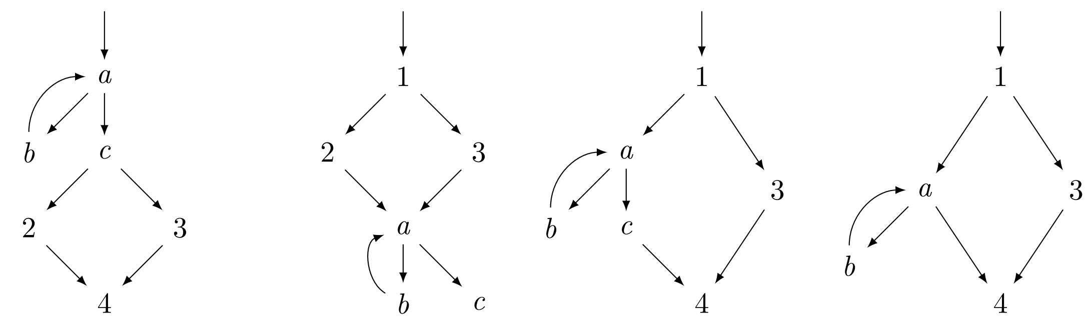
The third case (Figure 5.3c) is that of replacing an internal node of the graph. In our example, node 2 of graph G is replaced with graph H. There is an edge from the predecessor of node 2, which is node 1 in this case, to the root node of H (node a). There is another edge from the end node of H (node c), which is required to be unique, to the successor of node 2, which is node 4 in this case.

The final case, an example of which is shown in Figure 5.3d, is where control flow constructs occur at the end of one branch of a conditional. In our example, node 2 of graph G is replaced with graph H, as in the previous case, but the end node of H (node c) is identified with the successor of node 2 (node 4), and so it is not included in the graph. Thus, this represents the case in which no instructions occur inside the conditional branch after the while loop. Such instructions are represented by node c in Figure 5.3c, which is excluded in Figure 5.3d.

In general, we define node replacement using the formal definition below. This covers each of the four cases shown above. Note that the root node is the only node that may have no predecessors, since every node must be reachable from the root node, but there are some graphs, such as Figure 5.1e, where the root node does have a predecessor. The root node cannot be replaced in such graphs.

Definition 5.2.2 (Node Replacement). Given two rooted directed graphs G and H, we say G' is the graph formed by *replacing* a node n of G with H if one of the following cases holds:

- n has no predecessors in G, either H has only one end node or n has no successors in G, and
  - G' contains all the nodes of H and G, except n,
  - G' contains the edges of G and the edges of H except those going to or from n,



(a) root node replacement (b) end node replacement (c) internal node replacement (d) branch end replacement

Figure 5.3: Examples of the different cases of node replacement

  - G' contains edges from the end node of H to the successors of n in G (if any), and
  - the root node of G' is the root node of H;

- n has no successors in G, n is not the root node of G, and
  - G' contains all the nodes of H and G, except n,
  - G' contains the edges of G and the edges of H except those going to or from n,
  - G' contains edges from the predecessors of n in G to the root node of H, and
  - the root node of G' is the root node of G;
- H has a single end node, n is not the root node or an end node of G, and
  - G' contains all the nodes of H and G, except n,
  - G' contains the edges of G and the edges of H except those going to or from n,
  - G' contains edges from the predecessors of n in G to the root node of H,
  - G' contains edges from the end node of H to the successors of n in G, and
  - the root node of G' is the root node of G;
- n has a single successor in G, n is not the root node of G, H has a single end node, and
  - G' contains all the nodes of H and G, except n and the end node of H,
  - G' contains the edges of G and the edges of H except those going to or from n or the end node of H,
  - G' contains edges from the predecessors of the end node of H to the successor of n in G
  - G' contains edges from the predecessors of n in G to the root node of H, and
  - the root node of G' is the root node of G.

With node replacement defined, we can now finally define what we mean by a structured control flow graph in terms of node replacement and the structured graphs shown in Figure 5.1

> **Definition 5.2.3** (Structured Control Flow Graph)**.** If G is a rooted directed graph, we say G is a *structured control flow graph* if G is the trivial graph (the graph with a single node, which is also the root, and no edges) or if G can be created by starting with the trivial graph and performing a finite number of node replacements to replace nodes with graphs of the forms shown in Figure 5.1.

Before applying the strategy, it must be ensured that the control flow graph for each method is well-structured according to this definition.

Each method call must have at least one target (as determined by the rules given in Section 5.3.5), to allow method calls to be resolved. Each `invokestatic` and `invokespecial` instruction has exactly one target, so this property is always fulfilled for such method calls. For `invokevirtual` instructions, a method call only has no targets if the method in which the instruction occurs is unused or if the method is invoked on a null pointer (which is erroneous). Methods not used in the program should not be included in the parameters passed to CEE, matching icecap's behaviour of excluding such methods from the generated code.

Finally, we require that no method in the program recurses, either directly or indirectly. This is because recursion is not recommended in safety-critical applications because of the potential for unpredictable failure due to stack overflow, and it is not allowed in MISRA-C for that reason. Imposing this requirement allows us to handle methods individually when introducing their control flow, without considering circular dependencies between them.

The requirements discussed above are sufficient to ensure our compilation strategy can be applied to produce well-formed C code. If the generated code is additionally required to conform to MISRA-C, then integer overflow must be avoided in the input SCJ (Java) code. This is the only extra requirement on the Java code, in addition to those stated above, needed to ensure the generated C code conforms to MISRA-C. This additional requirement is needed due to the fact that we follow icecap's approach and compile addition in Java to addition in C, without applying special handling to overflows. The presence of such overflows would prevent the generated C code from being MISRA-C compliant, since signed integer overflow is undefined behaviour in C. We follow the approach of icecap on this, rather than generating checks for overflows in the C code, since it is important to follow the approach of a practical tool to ensure our strategy can be applied.

We also note that we follow icecap's approach in not applying synchronisation to static methods. Static synchronized methods must therefore not be used in code input to the strategy in order to ensure correct synchronisation behaviour. Singleton objects with synchronized methods may be used to achieve the same functionality as static synchronized methods.

We now proceed to describe each of the stages of the strategy in detail, beginning with the *Elimination of Program Counter* stage in the next section.

5.3 Elimination of Program Counter

The first stage eliminates pc from the state of each thread's process, $Thr(bc, cs, instCS, t)$, introducing the control flow constructs of C as a result. It is summarised by the following theorem.

Theorem 5.3.1 (Elimination of Program Counter)**.**

$$Thr(bc, cs, instCS, t) \sqsubseteq ThrCF_{bc,cs}(cs, t)$$

We act mainly upon the *Running* action of *Thr*; its loop is unrolled to introduce the control flow that follows each bytecode instruction. The aim is to get each method's bytecode instructions into a form in which the control flow, but not the data operations, are described using C constructs and, moreover, each path of execution (including every branch of the conditionals) ends in a return instruction or a loop. We refer to a method in this form as a *complete* method.

It is important to observe that it is possible to transform the bytecode instructions of every method so that they become complete. If we consider the control flow of a method beginning from that method's entry point, each bytecode instruction reached must either be a return instruction, or followed by another bytecode. If another bytecode follows the bytecode's execution, then it must be either a bytecode already considered, resulting in a loop, or one not already considered. Since there are finitely many bytecode instructions in a method, a loop or return must eventually be reached. Failure to do so would lead to an instruction beyond the end of the method, which is forbidden by the structural restrictions on Java bytecode that are checked during bytecode verification.

When a method is complete, it can be defined by a separate *Circus* action. When the code for all the methods has been separated out in this way, the choice of bytecode instruction using the program counter value can be removed and replaced with a choice over method identifiers. Thus dependency on the program counter can be completely removed, allowing it to be eliminated from the state of *Thr*.

The detailed description of the strategy for transforming *Thr* in this stage and achieving this elimination is provided by Algorithm 1. It begins at line 1 by expanding the *Circus* definitions of the bytecode instructions from the *bc* map into the *Running* action, pulling out the program counter updates so that they can be more easily manipulated. In line 2, simple sequential compositions, that is, those that do not involve handling loops or conditionals, are introduced. After that, for each method, its loops and conditionals are introduced in line 4. Afterwards, any complete methods are separated out, in line 5, and any method calls involving completed methods are resolved by sequencing the method call with the *Circus* action representing the method, in line 6.

This is repeated until all methods have been separated out, as indicated by the while loop in lines 3 to 7. The *MainThread* and *NotStarted* actions are then refined in line 8 to provide a choice over method identifiers, rather than *pc* values, thus removing all uses of *pc* from the interpreter. The *pc* component is then removed from the state in line 9 of the algorithm.

Each of the procedures used in Algorithm 1 is defined in a separate section in the sequel. Beforehand, we give a more detailed overview of the strategy using an example.

5.3.1 Running Example

We explain the strategy in detail with an example, the Java code for which is shown in Figure 5.4. Our example is based on the Trabb Pardo-Knuth algorithm [55], used for comparison of programming languages, since it includes a variety of programming constructs that provide a good test of the strategy. We have simplified the algorithm by removing the reading into an array, since our bytecode subset does not include array operations. Adding arrays makes the

Algorithm 1 Elimination of Program Counter

```
1: ExpandBytecode
2: IntroduceSequentialComposition
3: while ¬ AllMethodsSeparated do
4:     IntroduceLoopsAndConditionals
5:     SeparateCompleteMethods
6:     ResolveMethodCalls
7: end while
8: RefineMainActions
9: RemovePCFromState
```

```
 1 public class TPK extends AperiodicEventHandler {
 2
 3   public TPK(PriorityParameters priority,
 4              AperiodicParameters release,
 5              StorageParameters storage,
 6              ConfigurationParameters config) {
 7     super(priority, release, storage, config);
 8   }
 9
10   public void handleAsyncEvent() {
11     ConsoleConnection console = new ConsoleConnection(null);
12
13     InputStream input = console.openInputStream();
14     OutputStream output = console.openOutputStream();
15
16     for(int i = 0; i <= 10; i = i + 1) {
17       int y = f(input.read());
18
19       if (y > 400) {
20         output.write(0);
21       } else {
22         output.write(y);
23       }
24     }
25   }
26
27   public static int f(int x){
28     return x + x + x + 5;
29   }
30
31 }
```

Figure 5.4: Our example program

example much longer, while not giving any interesting insight into our compilation strategy. As previously explained, extending the bytecode set considered to deal with arrays is not difficult.

We have also written the example as an SCJ program, with the algorithm as the body of an aperiodic event handler, `TPK`, one or more instances of which can be registered as part of a mission and released during mission execution. As already mentioned, each release of the handler causes its `handleAsyncEvent()` method to be executed. This method creates an instance of a `ConsoleConnection` (line 11), which is the only standard input/output connection required by SCJ. Instances of `InputStream` and `OutputStream` are then obtained from the

$$
\begin{array}{|l}
\mathit{TPK} : \mathit{Class} \\
\hline
\mathit{TPK} = \llparenthesis \\
\quad \mathit{constantPool} == \{ \\
\qquad 1 \mapsto \mathit{ClassRef}\ \mathit{TPKClassID}, \\
\qquad 3 \mapsto \mathit{ClassRef}\ \mathit{AperiodicEventHandlerClassID}, \\
\qquad 8 \mapsto \mathit{MethodRef}\ \mathit{AperiodicEventHandlerClassID}\ \mathit{APEHinit}, \\
\qquad 27 \mapsto \mathit{ClassRef}\ \mathit{ConsoleConnectionClassID}, \\
\qquad 29 \mapsto \mathit{MethodRef}\ \mathit{ConsoleConnectionClassID}\ \mathit{CCinit}, \\
\qquad 32 \mapsto \mathit{MethodRef}\ \mathit{ConsoleConnectionClassID}\ \mathit{openInputStream}, \\
\qquad 36 \mapsto \mathit{MethodRef}\ \mathit{ConsoleConnectionClassID}\ \mathit{openOutputStream}, \\
\qquad 40 \mapsto \mathit{MethodRef}\ \mathit{InputStreamClassID}\ \mathit{read}, \\
\qquad 41 \mapsto \mathit{ClassRef}\ \mathit{InputStreamClassID}, \\
\qquad 46 \mapsto \mathit{MethodRef}\ \mathit{TPKClassID}\ f, \\
\qquad 50 \mapsto \mathit{MethodRef}\ \mathit{OutputStreamClassID}\ \mathit{write}, \\
\qquad 51 \mapsto \mathit{ClassRef}\ \mathit{OutputStreamClassID} \\
\quad \}, \\
\quad \mathit{this} == 1, \\
\quad \mathit{super} == 3, \\
\quad \mathit{interfaces} == \{\}, \\
\quad \mathit{methodEntry} == \{ \\
\qquad f \mapsto 43, \\
\qquad \mathit{handleAsyncEvent} \mapsto 7, \\
\qquad \mathit{APEHinit} \mapsto 0, \\
\quad \}, \\
\quad \mathit{methodEnd} == \{ \\
\qquad f \mapsto 50, \\
\qquad \mathit{handleAsyncEvent} \mapsto 42, \\
\qquad \mathit{APEHinit} \mapsto 6 \\
\quad \}, \\
\quad \mathit{methodLocals} == \{ \\
\qquad f \mapsto 1, \\
\qquad \mathit{handleAsyncEvent} \mapsto 6, \\
\qquad \mathit{APEHinit} \mapsto 5, \\
\quad \}, \\
\quad \mathit{methodStackSize} == \{ \\
\qquad f \mapsto 2, \\
\qquad \mathit{handleAsyncEvent} \mapsto 3, \\
\qquad \mathit{APEHinit} \mapsto 5, \\
\quad \}, \\
\quad \mathit{staticMethods} == \{f\} \\
\quad \mathit{fields} == \{\}, \\
\quad \mathit{staticFields} == \{\} \\
\rrparenthesis
\end{array}
$$

$$
\begin{array}{|l}
\mathit{cs} : \mathit{ClassID} \mathrel{+\!\!\!\!\to} \mathit{Class} \\
\hline
\mathit{cs} = \{ \\
\quad \mathit{TPKClassID} \mapsto \mathit{TPK}, \\
\quad \mathit{AperiodicEventHandlerClassID} \mapsto \mathit{AperiodicEventHandler}, \\
\quad \mathit{ManagedEventHandlerClassID} \mapsto \mathit{ManagedEventHandler}, \\
\quad \ldots \\
\}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{bc} : \mathit{ProgramAddress} \mathrel{+\!\!\!\!\to} \mathit{Bytecode} \\
\hline
\mathit{bc} = \{ \\
\quad 0 \mapsto \mathit{aload}\,0, \\
\quad 1 \mapsto \mathit{aload}\,1, \\
\quad 2 \mapsto \mathit{aload}\,2, \\
\quad 3 \mapsto \mathit{aload}\,3, \\
\quad 4 \mapsto \mathit{aload}\,4, \\
\quad 5 \mapsto \mathit{invokespecial}\,8, \\
\quad 6 \mapsto \mathit{return}, \\
\quad 7 \mapsto \mathit{new}\,27, \\
\quad 8 \mapsto \mathit{dup}, \\
\quad 9 \mapsto \mathit{aconst\_null}, \\
\quad 10 \mapsto \mathit{invokespecial}\,29, \\
\quad 11 \mapsto \mathit{astore}\,1, \\
\quad 12 \mapsto \mathit{aload}\,1, \\
\quad 13 \mapsto \mathit{invokevirtual}\,32, \\
\quad 14 \mapsto \mathit{astore}\,2, \\
\quad 15 \mapsto \mathit{aload}\,1, \\
\quad 16 \mapsto \mathit{invokevirtual}\,36, \\
\quad 17 \mapsto \mathit{astore}\,3, \\
\quad 18 \mapsto \mathit{iconst}\,0, \\
\quad 19 \mapsto \mathit{astore}\,4, \\
\quad 20 \mapsto \mathit{goto}\,19, \\
\quad 21 \mapsto \mathit{aload}\,2, \\
\quad 22 \mapsto \mathit{invokevirtual}\,40, \\
\quad 23 \mapsto \mathit{invokestatic}\,46, \\
\quad 24 \mapsto \mathit{astore}\,5, \\
\quad 25 \mapsto \mathit{aload}\,5, \\
\quad 26 \mapsto \mathit{iconst}\,400, \\
\quad 27 \mapsto \mathit{if\_icmple}\,5, \\
\quad 28 \mapsto \mathit{aload}\,3, \\
\quad 29 \mapsto \mathit{iconst}\,0, \\
\quad 30 \mapsto \mathit{invokevirtual}\,50, \\
\quad 31 \mapsto \mathit{goto}\,4, \\
\quad 32 \mapsto \mathit{aload}\,3, \\
\quad 33 \mapsto \mathit{aload}\,5, \\
\quad 34 \mapsto \mathit{invokevirtual}\,50, \\
\quad 35 \mapsto \mathit{aload}\,4, \\
\quad 36 \mapsto \mathit{iconst}\,1, \\
\quad 37 \mapsto \mathit{iadd}, \\
\quad 38 \mapsto \mathit{astore}\,4, \\
\quad 39 \mapsto \mathit{aload}\,4, \\
\quad 40 \mapsto \mathit{iconst}\,10, \\
\quad 41 \mapsto \mathit{if\_icmple}\,(-\,20), \\
\quad 42 \mapsto \mathit{return}, \\
\quad 43 \mapsto \mathit{aload}\,0, \\
\quad 44 \mapsto \mathit{aload}\,0, \\
\quad 45 \mapsto \mathit{iadd}, \\
\quad 46 \mapsto \mathit{aload}\,0, \\
\quad 47 \mapsto \mathit{iadd}, \\
\quad 48 \mapsto \mathit{iconst}\,5, \\
\quad 49 \mapsto \mathit{iadd}, \\
\quad 50 \mapsto \mathit{areturn}, \\
\quad \ldots \\
\}
\end{array}
$$

Figure 5.5: The *Circus* code corresponding to our example program

```
AperiodicEventHandler : Class
─────────────────────────────
AperiodicEventHandler = ⦇
  constantPool == {
    1 ↦ ClassRef AperiodicEventHandlerClassID,
    3 ↦ ClassRef ManagedEventHandlerClassID,
    . . .
  },
  this == 1,
  super == 3,
  interfaces == {},
  methodEntry == {· · ·},
  methodEnd == {· · ·},
  methodLocals == {· · ·},
  methodStackSize == {· · ·},
  fields == {},
  staticFields == {}
⦈
```

```
ManagedEventHandler : Class
───────────────────────────
ManagedEventHandler = ⦇
  constantPool == {
    1 ↦ ClassRef ManagedEventHandlerClassID,
    3 ↦ ClassRef BoundAsyncEventHandlerClassID,
    5 ↦ ClassRef ManagedSchedulableClassID,
    . . .
  },
  this == 1,
  super == 3,
  interfaces == {5},
  methodEntry == {· · ·},
  methodEnd == {· · ·},
  methodLocals == {· · ·},
  methodStackSize == {· · ·},
  fields == {
    threadID,
    backingStoreSpace,
    allocAreaSpace,
    stackSize
  },
  staticFields == {}
⦈
```

Figure 5.6: The *Class* structures for `AperiodicEventHandler` and `ManagedEventHandler`

`ConsoleConnection` (lines 13 and 14).

After the input and output streams have been obtained, we enter a for loop (line 16) in which an integer is read from the `InputStream`, a static method `f()` is applied to it, and the result is output if it is less than 400, otherwise, 0 is output. The method `f()` takes an integer as input, multiplies it by 3 and adds 5 to it.

The `TPK` class is part of a larger program that includes other classes, including a `Safelet`, a `MissionSequencer`, a `Mission`, and the classes that make up the SCJ API. We omit a presentation of these classes, though it should be noted that they are part of the complete example. For compilation, they need to go through a similar refinement to that we illustrate for the *TPK* class. This adds little complexity to the strategy since the bytecode array is acted upon consistently for all classes, and the current class of a given bytecode instruction can always be determined from its address in the array.

The Java code must be run through a Java compiler to generate the corresponding bytecode, which then defines the *bc* and *cs* constants of our model. Their values for our example are shown in Figure 5.5, along with the *TPK* class information. While most of the compilation of the methods of `TPK` depends only on the data in the *TPK* class information, the object data for instances of `TPK` includes fields from its superclasses. In particular, the fields for `TPK` are contributed by the `AperiodicEventHandler` and `ManagedEventHandler` classes (since the superclasses of `ManagedEventHandler` do not contribute any fields), whose *Class* data structures are presented in Figure 5.6. We omit the information not involved in determining the fields of those classes. The generation of object structures from this field information is discussed in more detail in Section 5.5.

Applying the bytecode expansion on line 1 of Algorithm 1 yields the *Running* action shown in Figure 5.7. This step copies *HandleInstruction* into *Running*, and converts it to a choice of actions based on the value of the program counter, *pc*, mirroring the contents of the *bc* map for each value.

```
$Running \mathrel{\widehat=}$
  $\mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip}$
  $[]\ frameStack \neq \varnothing \longrightarrow$
    $\mathbf{if}\ pc = 0 \longrightarrow HandleAloadEPC(0) \mathbin{;}\ pc := 1$
     $[]\ pc = 1 \longrightarrow HandleAloadEPC(1) \mathbin{;}\ pc := 2$
     $[]\ pc = 2 \longrightarrow HandleAloadEPC(2) \mathbin{;}\ pc := 3$
     $[]\ pc = 3 \longrightarrow HandleAloadEPC(3) \mathbin{;}\ pc := 4$
     $[]\ pc = 4 \longrightarrow HandleAloadEPC(4) \mathbin{;}\ pc := 5$
     $[]\ pc = 5 \longrightarrow \{pc = 5\} \mathbin{;}\ HandleInvokespecialEPC(8)$
     $[]\ pc = 6 \longrightarrow HandleReturnEPC$
     $[]\ pc = 7 \longrightarrow HandleNewEPC(27) \mathbin{;}\ pc := 8$
     $[]\ pc = 8 \longrightarrow HandleDupEPC \mathbin{;}\ pc := 9$
     $[]\ pc = 9 \longrightarrow HandleAconst\_nullEPC \mathbin{;}\ pc := 10$
     $[]\ pc = 10 \longrightarrow \{pc = 10\};$
       $HandleInvokespecialEPC(29)$
     $[]\ pc = 11 \longrightarrow HandleAstoreEPC(1) \mathbin{;}\ pc := 12$
     $[]\ pc = 12 \longrightarrow HandleAloadEPC(1) \mathbin{;}\ pc := 13$
     $[]\ pc = 13 \longrightarrow \{pc = 13\};$
       $HandleInvokevirtualEPC(32)$
     $[]\ pc = 14 \longrightarrow HandleAstoreEPC(2) \mathbin{;}\ pc := 15$
     $[]\ pc = 15 \longrightarrow HandleAloadEPC(1) \mathbin{;}\ pc := 16$
     $[]\ pc = 16 \longrightarrow \{pc = 16\};$
       $HandleInvokevirtualEPC(36)$
     $[]\ pc = 17 \longrightarrow HandleAstoreEPC(3) \mathbin{;}\ pc := 18$
     $[]\ pc = 18 \longrightarrow HandleIconstEPC(0) \mathbin{;}\ pc := 19$
     $[]\ pc = 19 \longrightarrow HandleAstoreEPC(4) \mathbin{;}\ pc := 20$
     $[]\ pc = 20 \longrightarrow pc := 39$
     $[]\ pc = 21 \longrightarrow HandleAloadEPC(2) \mathbin{;}\ pc := 22$
     $[]\ pc = 22 \longrightarrow \{pc = 22\};$
       $HandleInvokevirtualEPC(40)$
     $[]\ pc = 23 \longrightarrow \{pc = 23\};$
       $HandleInvokestaticEPC(46)$
     $[]\ pc = 24 \longrightarrow HandleAstoreEPC(5) \mathbin{;}\ pc := 25$
     $[]\ pc = 25 \longrightarrow HandleAloadEPC(5) \mathbin{;}\ pc := 26$
     $[]\ pc = 26 \longrightarrow HandleIconstEPC(400) \mathbin{;}\ pc := 27$
     $[]\ pc = 27 \longrightarrow \mathbf{var}\ value1, value2 : Word \bullet$
       $\left(InterpreterPopEPC[value2!/value!]\right);$
       $\left(InterpreterPopEPC[value1!/value!]\right);$
       $pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ 32\ \mathbf{else}\ 28$
     $[]\ pc = 28 \longrightarrow HandleAloadEPC(3) \mathbin{;}\ pc := 29$
     $[]\ pc = 29 \longrightarrow HandleIconstEPC(0) \mathbin{;}\ pc := 30$
     $[]\ pc = 30 \longrightarrow \{pc = 30\};$
       $HandleInvokevirtualEPC(50)$
     $[]\ pc = 31 \longrightarrow pc := 35$
     $[]\ pc = 32 \longrightarrow HandleAloadEPC(3) \mathbin{;}\ pc := 33$
     $[]\ pc = 33 \longrightarrow HandleAloadEPC(5) \mathbin{;}\ pc := 34$
     $[]\ pc = 34 \longrightarrow \{pc = 34\};$
       $HandleInvokevirtualEPC(50)$
     $[]\ pc = 35 \longrightarrow HandleAloadEPC(4) \mathbin{;}\ pc := 36$
     $[]\ pc = 36 \longrightarrow HandleIconstEPC(1) \mathbin{;}\ pc := 37$
     $[]\ pc = 37 \longrightarrow HandleIaddEPC \mathbin{;}\ pc := 38$
     $[]\ pc = 38 \longrightarrow HandleAstoreEPC(4) \mathbin{;}\ pc := 39$
     $[]\ pc = 39 \longrightarrow HandleAloadEPC(4) \mathbin{;}\ pc := 40$
     $[]\ pc = 40 \longrightarrow HandleIconstEPC(10) \mathbin{;}\ pc := 41$
     $[]\ pc = 41 \longrightarrow \mathbf{var}\ value1, value2 : Word \bullet$
       $\left(InterpreterPopEPC[value2!/value!]\right);$
       $\left(InterpreterPopEPC[value1!/value!]\right);$
       $pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ 21\ \mathbf{else}\ 42$
     $[]\ pc = 42 \longrightarrow HandleReturnEPC$
     $[]\ pc = 43 \longrightarrow HandleAloadEPC(0) \mathbin{;}\ pc := 44$
     $[]\ pc = 44 \longrightarrow HandleAloadEPC(0) \mathbin{;}\ pc := 45$
     $[]\ pc = 45 \longrightarrow HandleIaddEPC \mathbin{;}\ pc := 46$
     $[]\ pc = 46 \longrightarrow HandleAloadEPC(0) \mathbin{;}\ pc := 47$
     $[]\ pc = 47 \longrightarrow HandleIaddEPC \mathbin{;}\ pc := 48$
     $[]\ pc = 48 \longrightarrow HandleIconstEPC(5) \mathbin{;}\ pc := 49$
     $[]\ pc = 49 \longrightarrow HandleIaddEPC \mathbin{;}\ pc := 50$
     $[]\ pc = 50 \longrightarrow HandleAreturnEPC$
     $\ldots$
    $\mathbf{fi} \mathbin{;}\ Poll \mathbin{;}\ Running$
  $\mathbf{fi}$
```

Figure 5.7: The *Running* action after bytecode expansion

```
$Running \mathrel{\widehat=}$
  $\mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip}$
  $[\!]\ frameStack \neq \varnothing \longrightarrow$
    $\mathbf{if}\ pc = 0 \longrightarrow HandleAloadEPC(0) \mathbin{;} pc := 1 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(1) \mathbin{;} pc := 2 \mathbin{;} Poll \mathbin{;}$
      $HandleAloadEPC(2) \mathbin{;} pc := 3 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(4) \mathbin{;} pc := 5 \mathbin{;} Poll \mathbin{;}$
      $\{pc = 5\} \mathbin{;} HandleInvokespecialEPC(8)$
    $\ldots$
    $[\!]\ pc = 6 \longrightarrow HandleReturnEPC$
    $[\!]\ pc = 7 \longrightarrow HandleNewEPC(27) \mathbin{;} pc := 8 \mathbin{;} Poll \mathbin{;} HandleDupEPC \mathbin{;} pc := 9 \mathbin{;} Poll \mathbin{;}$
      $HandleAconst\_nullEPC \mathbin{;} pc := 10 \mathbin{;} Poll \mathbin{;} \{pc = 10\} \mathbin{;} HandleInvokespecialEPC(29)$
    $\ldots$
    $[\!]\ pc = 11 \longrightarrow HandleAstoreEPC(1) \mathbin{;} pc := 12 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(1) \mathbin{;} pc := 13 \mathbin{;} Poll \mathbin{;}$
      $\{pc = 13\} \mathbin{;} HandleInvokevirtualEPC(32)$
    $\ldots$
    $[\!]\ pc = 14 \longrightarrow HandleAstoreEPC(2) \mathbin{;} pc := 15 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(1) \mathbin{;} pc := 16 \mathbin{;} Poll \mathbin{;}$
      $\{pc = 16\} \mathbin{;} HandleInvokevirtualEPC(36)$
    $\ldots$
    $[\!]\ pc = 17 \longrightarrow HandleAstoreEPC(3) \mathbin{;} pc := 18 \mathbin{;} Poll \mathbin{;} HandleIconstEPC(0) \mathbin{;} pc := 19 \mathbin{;} Poll \mathbin{;}$
      $HandleAstoreEPC(4) \mathbin{;} pc := 20 \mathbin{;} Poll \mathbin{;} pc := 39$
    $\ldots$
    $[\!]\ pc = 21 \longrightarrow HandleAloadEPC(2) \mathbin{;} pc := 22 \mathbin{;} Poll \mathbin{;} \{pc = 22\} \mathbin{;} HandleInvokevirtualEPC(40)$
    $\ldots$
    $[\!]\ pc = 23 \longrightarrow HandleInvokestaticEPC(46)$
    $[\!]\ pc = 24 \longrightarrow HandleAstoreEPC(5) \mathbin{;} pc := 25 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(5) \mathbin{;} pc := 26 \mathbin{;} Poll \mathbin{;}$
      $HandleIconstEPC(400) \mathbin{;} pc := 27 \mathbin{;} Poll \mathbin{;} \mathbf{var}\ value1, value2 : Word \bullet$
      $\big(InterpreterPopEPC[value2!/value!]\big) \mathbin{;} \big(InterpreterPopEPC[value1!/value!]\big) \mathbin{;}$
      $pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ 32\ \mathbf{else}\ 28$
    $[\!]\ pc = 28 \longrightarrow HandleAloadEPC(3) \mathbin{;} pc := 29 \mathbin{;} Poll \mathbin{;} HandleIconstEPC(0) \mathbin{;} pc := 30 \mathbin{;} Poll \mathbin{;}$
      $\{pc = 30\} \mathbin{;} HandleInvokevirtualEPC(50)$
    $\ldots$
    $[\!]\ pc = 31 \longrightarrow pc := 35$
    $[\!]\ pc = 32 \longrightarrow HandleAloadEPC(3) \mathbin{;} pc := 33 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(5) \mathbin{;}$
      $pc := 34 \mathbin{;} Poll \mathbin{;} \{pc = 34\} \mathbin{;} HandleInvokevirtualEPC(50)$
    $\ldots$
    $[\!]\ pc = 35 \longrightarrow HandleAloadEPC(4) \mathbin{;} pc := 36 \mathbin{;} Poll \mathbin{;} HandleIconstEPC(1) \mathbin{;} pc := 37 \mathbin{;} Poll \mathbin{;}$
      $HandleIaddEPC \mathbin{;} pc := 38 \mathbin{;} Poll \mathbin{;} HandleAstoreEPC(4) \mathbin{;} pc := 39$
    $\ldots$
    $[\!]\ pc = 39 \longrightarrow HandleAloadEPC(4) \mathbin{;} pc := 40 \mathbin{;} Poll \mathbin{;} HandleIconstEPC(10) \mathbin{;} pc := 41 \mathbin{;} Poll \mathbin{;}$
      $\mathbf{var}\ value1, value2 : Word \bullet \big(InterpreterPopEPC[value2!/value!]\big) \mathbin{;}$
      $\big(InterpreterPopEPC[value1!/value!]\big) \mathbin{;} pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ 21\ \mathbf{else}\ 42$
    $\ldots$
    $[\!]\ pc = 42 \longrightarrow HandleReturnEPC$
    $[\!]\ pc = 43 \longrightarrow HandleAloadEPC(0) \mathbin{;} pc := 44 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(0) \mathbin{;} pc := 45 \mathbin{;} Poll \mathbin{;}$
      $HandleIaddEPC \mathbin{;} pc := 46 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(0) \mathbin{;} pc := 47 \mathbin{;} Poll \mathbin{;} HandleIaddEPC \mathbin{;}$
      $pc := 48 \mathbin{;} Poll \mathbin{;} HandleIaddEPC \mathbin{;} pc := 50 \mathbin{;} Poll \mathbin{;} HandleAreturnEPC$
    $\ldots$
    $\mathbf{fi} \mathbin{;} Poll \mathbin{;} Running$
  $\mathbf{fi}$
```

Figure 5.8: The *Running* action after forward sequence introduction

The actions that make up *HandleInstruction* are also replaced with actions that incorporate instruction parameters from the *bc* map, and have *pc* updates separated from stack updates. This can be seen in Figure 5.7, where, for instance, in the $pc = 0$ case, $aload\ 0$ has been converted to $HandleAloadEPC(0) \mathbin{;} pc := 1$, with the parameter, 0, to the bytecode instruction becoming a parameter of the new instruction handling action *HandleAloadEPC*, and the update to *pc* placed after the data operation.

The reason for making parameters of the bytecode instructions into parameters of the handling actions is to remove the need to reference the bytecode instructions in the *bc* map, as that involves use of the *pc* value, which we seek to remove in this stage. This also has the benefit of fully incorporating *bc* into the *Thr* process, ensuring all the information required to introduce C code constructs is available directly in *Circus*. This makes stating compilation laws simpler, and is described in more detail in Section 5.3.2, where we define the EXPANDBYTECODE procedure.

On line 2 of Algorithm 1, sequential composition is introduced for instructions that do not affect the sequential flow of the program. Such instructions are identified by considering the control flow graph of the program and locating nodes with a single outgoing edge going to a target node with exactly one incoming edge. The introduction of sequential composition is performed by unrolling the loop in *Running* to introduce the control flow following each of these instructions. This causes the instruction to be sequentially composed with the next instruction, with *Poll* inbetween to allow for thread switches between instructions. This is performed exhaustively to get the code in the form shown in Figure 5.8, where the choice over *pc* has sequences of instructions collected together at the point where they start, up to the point at which a more complex control flow (such as a method call, conditional or a loop) occurs. In this figure, and the other figures in this chapter, we omit the branches of the choice in *Running* that cannot be reached from the start of a method, since the instructions in those branches are collected into other branches. The introduction of sequential composition is described in more detail in Section 5.3.3, where we define the INTRODUCESEQUENTIALCOMPOSITION procedure.

Handling the remaining constructs requires consideration of dependencies between methods to ensure method calls can be resolved correctly. We say a method call is *resolved* when the method invocation bytecode has been placed in sequential composition with a call to a *Circus* action containing the body of the method being invoked, which is then followed by the sequence of instructions that occurs after the invocation bytecode in the calling method. After a method call has been resolved, it no longer breaks up the sequence of instructions it occurs in.

Since we have the bytecode instructions of all the methods needed, we can always resolve the call of a complete method, provided that method has already been split into its own *Circus* action. To obtain a complete method, we first perform loop and conditional introduction upon the method. Since introducing loops and conditionals requires unbroken sequences of instructions that form the bodies of the loops and the branches of conditionals, introduction of loops and conditionals can only be performed on methods that have no unresolved method calls.

In our example, `handleAsyncEvent()` is the only method that needs loops and conditionals introducing but, since it also contains method calls that break up the body of a loop, we must wait until its method calls have been resolved before introducing loops and conditionals. For this reason, we perform method call resolution, and loop and conditional introduction repeatedly until all method calls are resolved and the resulting complete methods have all been separated out. This is expressed in Algorithm 1 by the while loop in lines 3 to 7.

Introduction of loops and conditionals to the body of a method with no unresolved method calls occurs on line 4 of the algorithm. To introduce loops and conditionals we consider the control flow graph of the method again, though it is now much simpler than the control flow graph used for sequence introduction, since straight sequences of instructions have already been combined together. Patterns representing conditionals and loops are then identified using the control flow graph and the corresponding constructs are introduced. As loops and conditionals are introduced, nodes in the control flow graph are merged until the graph consists of a single node, which is the starting point of the method, containing the complete method body.

The result of introducing loops and conditionals in `handleAsyncEvent()` after method call resolution is shown in Figure 5.9. The process of introducing loops and conditionals is described in more detail in Section 5.3.4, where we define the INTRODUCELOOPSANDCONDITIONALS procedure.

```
$Running \mathrel{\widehat{=}}$
 $\mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip}$
 $[\!]\ frameStack \neq \varnothing \longrightarrow$
  $\mathbf{if}\ pc = 0 \longrightarrow HandleAloadEPC(0) ; pc := 1 ; Poll ; HandleAloadEPC(1) ; pc := 2 ; Poll ;$
   $HandleAloadEPC(2) ; pc := 3 ; Poll ; HandleAloadEPC(3) ; pc := 4 ; Poll ; HandleAloadEPC(4) ; pc := 5 ;$
   $Poll ; (\mathbf{var}\ poppedArgs : \operatorname{seq} Word \bullet \big(\exists\, argsToPop? == 6 \bullet InterpreterStackFrameInvokeEPC\big) ;$
   $\big(InterpreterNewStackFrame[AperiodicEventHandler/class?, APEHinit/methodID?, poppedArgs/methodArgs?]\big) ;$
   $Poll ; AperiodicEventHandler\_APEHinit) ; pc := 6 ; Poll ; HandleReturnEPC$
  $\ldots$
  $[\!]\ pc = 7 \longrightarrow HandleNewEPC(27) ; pc := 8 ; Poll ; HandleDupEPC ; pc := 9 ; Poll ; HandleAconst\_nullEPC ;$
   $pc := 10 ; Poll ; (\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\, argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\big) ;$
   $\big(InterpreterNewStackFrame[ConsoleConnection/class?, CCinit/methodID?]\big) ; Poll ;$
   $ConsoleConnection\_CCinit) ; pc := 11 ; Poll ; HandleAstoreEPC(1) ; pc := 12 ; Poll ;$
   $HandleAloadEPC(1) ; pc := 13 ; Poll ; (\mathbf{var}\ poppedArgs : Word \bullet$
   $\big(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\big) ; getClassIDOf!head\ poppedArgs?cid \longrightarrow$
   $\mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
    $\big(InterpreterNewStackFrame[ConsoleConnection/class?, openInputStream/methodID?]\big) ;$
    $Poll ; ConsoleConnection\_openInputStream$
   $\mathbf{fi}) ; pc := 14 ; Poll ; HandleAstoreEPC(2) ; pc := 15 ; Poll ; HandleAloadEPC(1) ; pc := 16 ; Poll ;$
   $(\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\big) ;$
   $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
    $\big(InterpreterNewStackFrame[ConsoleConnection/class?, openOutputStream/methodID?]\big) ; Poll ;$
    $ConsoleConnection\_openOutputStream$
   $\mathbf{fi}) ; pc := 17 ; Poll ; HandleAstoreEPC(3) ; pc := 18 ; Poll ; HandleIconstEPC(0) ; pc := 19 ; Poll ;$
   $HandleAstoreEPC(4) ; pc := 20 ; Poll ; pc := 39 ; Poll ; (\mu\, Y \bullet$
    $HandleAloadEPC(4) ; pc := 40 ; Poll ; HandleIconstEPC(10) ; pc := 41 ; Poll ;$
    $(\mathbf{var}\ value1, value2 : Word \bullet \big(InterpreterPopEPC[value2!/value!]\big) ; \big(InterpreterPopEPC[value1!/value!]\big) ;$
    $\mathbf{if}\ value1 \leq value2 \longrightarrow pc := 21 ; Poll ; HandleAloadEPC(2) ; pc := 22 ; Poll ;$
     $(\mathbf{var}\ poppedArgs : \operatorname{seq} Word \bullet \big(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\big) ;$
     $getClassIDOf!(head\ poppedArgs)?cid \longrightarrow \mathbf{if}\ cid = ConsoleInputClassID \longrightarrow$
      $\big(InterpreterNewStackFrame[ConsoleInput/class?, read/methodID?, poppedArgs/methodArgs?]\big) ;$
      $Poll ; ConsoleInput\_read$
     $\mathbf{fi}) ; pc := 23 ; Poll ; (\mathbf{var}\ poppedArgs : \operatorname{seq} Word \bullet \big(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\big) ;$
     $\big(InterpreterNewStackFrame[TPK/class?, f/methodID?, poppedArgs/methodArgs?]\big) ; Poll ; TPK\_f) ;$
     $pc := 24 ; Poll ; HandleAstoreEPC(5) ; pc := 25 ; Poll ; HandleAloadEPC(5) ; pc := 26 ;$
     $HandleIconstEPC(400) ; pc := 27 ; Poll ; (\mathbf{var}\ value1, value2 : Word \bullet$
     $\big(InterpreterPopEPC[value2!/value!]\big) ; \big(InterpreterPopEPC[value1!/value!]\big) ;$
     $\mathbf{if}\ value1 \leq value2 \longrightarrow pc := 32 ; HandleAloadEPC(3) ; pc := 33 ; Poll ; HandleAloadEPC(5) ; pc := 34 ;$
      $Poll ; (\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\, argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\big) ;$
      $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
       $\big(InterpreterNewStackFrame[ConsoleOutput/class?, write/methodID?]\big) ; Poll ; ConsoleOutput\_write$
      $\mathbf{fi})$
     $[\!]\ value1 > value2 \longrightarrow pc := 28 ; HandleAloadEPC(3) ; pc := 29 ; Poll ; HandleIconstEPC(0) ; pc := 30 ;$
      $Poll ; (\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\, argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\big) ;$
      $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
       $\big(InterpreterNewStackFrame[ConsoleOutput/class?, write/methodID?]\big) ; Poll ; ConsoleOutput\_write$
      $\mathbf{fi}) ; pc := 31 ; Poll$
     $\mathbf{fi}) ; pc := 35 ; Poll ; HandleAloadEPC(4) ; pc := 36 ; Poll ; HandleIconstEPC(1) ; pc := 37 ; Poll ;$
     $HandleIaddEPC ; pc := 38 ; Poll ; HandleAstoreEPC(4) ; pc := 39 ; Poll ; Y$
    $[\!]\ value1 > value2 \longrightarrow \mathbf{Skip}$
    $\mathbf{fi})) ; pc := 42 ; Poll ; HandleReturnEPC$
  $\ldots$
  $[\!]\ pc = 43 \longrightarrow TPK\_f$
  $\ldots$
  $\mathbf{fi}) ; Poll ; Running$
 $\mathbf{fi}$
```

Figure 5.9: The *Running* action after loop and conditional introduction

$TPK\_f \mathrel{\widehat=} HandleAloadEPC(0) \mathbin{;} pc := 44 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(0) \mathbin{;} pc := 45 \mathbin{;} Poll;$
$\quad HandleIaddEPC \mathbin{;} pc := 46 \mathbin{;} Poll \mathbin{;} HandleAloadEPC(0) \mathbin{;} pc := 47 \mathbin{;} Poll;$
$\quad HandleIaddEPC \mathbin{;} pc := 48 \mathbin{;} Poll \mathbin{;} HandleIconstEPC(5) \mathbin{;} pc := 49 \mathbin{;} Poll;$
$HandleIaddEPC \mathbin{;} pc := 50 \mathbin{;} Poll \mathbin{;} HandleAreturnEPC$

Figure 5.10: The TPK_f method action after it has been separated

After loops and conditionals have been introduced, methods that are then complete can be copied into separate actions. This occurs in line 5 of Algorithm 1. It is done with a simple application of the copy rule, replacing the actions at the entry points of the split methods with references to newly created method actions. This can be seen in Figure 5.10, where the TPK_f action has been created by copying the sequence of actions for the `f()` method of `TPK` from the $pc = 43$ case of Figure 5.8. The $pc = 43$ branch itself is replaced with a reference to this TPK_f action, as can be seen in Figure 5.9. As this step is relatively simple, we do not explain it in a separate section.

Calls to methods with separate actions can then be resolved, sequencing the method invocation instruction with a call to the *Circus* action representing its body and the instructions following the method call. This occurs on line 6 of the algorithm, and its result can be seen in Figure 5.11, which shows our example after method call resolution has been applied.

The target of each method call can be determined from the parameter to the method invocation instruction. This parameter is an index into the constant pool of the current class that points to a reference to the method being called. The correct current class for each bytecode instruction is always known, since the information on the method entries and ends is contained in the class information, and there is a one-to-one mapping between classes and blocks of bytecode instructions that form methods. After the target of the method call has been determined, the invocation instruction can be sequenced with a call to the corresponding *Circus* action.

An example of a resolved method call is the call to TPK_f at $pc = 23$, in the sequence of actions beginning at $pc = 21$ in Figure 5.11. This comes from resolving the method invocation instruction $invokestatic\,46$. As can be seen from Figure 5.5, the constant pool index 46 corresponds to the method identifier for the method `f()` of `TPK`. The sequence of instructions corresponding to this method is in an action TPK_f, created in the previous step, on line 5 of Algorithm 1.

The semantics for the invocation instruction is expanded to instantiate the data operations it contains. These are then sequenced with the method action TPK_f, with the $Poll$ action inbetween (to allow thread switches before the first instruction of the called method). The instructions following the method call are sequenced after it, with another $Poll$ action (to allow thread switches following the return from the method). Method call resolution is described in more detail in Section 5.3.5, where we define the SeparateCompleteMethods and ResolveMethodCalls procedures.

As mentioned previously, these steps are then repeated, in the loop beginning at line 3 of Algorithm 1 to introduce the loops and conditionals in methods that have unresolved method calls in the middle of loops and conditionals. Afterwards, those methods can be separated out and this loop, conditional and method resolution repeated until every method has been separated out in this way. This always terminates, since we do not allow recursion, and so there are no loops in the dependencies between methods.

The *Running* action of our example at the end of the loop in Algorithm 1, when all loops and

```
$Running \mathrel{\widehat=}$
  $\mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip}$
  $[]\ frameStack \neq \varnothing \longrightarrow$
    $\mathbf{if}\ pc = 0 \longrightarrow HandleAloadEPC(0)\ ;\ pc := 1\ ;\ Poll\ ;\ HandleAloadEPC(1)\ ;\ pc := 2\ ;\ Poll;$
      $HandleAloadEPC(2)\ ;\ pc := 3\ ;\ Poll\ ;\ HandleAloadEPC(3)\ ;\ pc := 4\ ;\ Poll\ ;\ HandleAloadEPC(4)\ ;\ pc := 5;$
      $Poll\ ;\ (\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \Big(\exists\ argsToPop? == 6 \bullet InterpreterStackFrameInvokeEPC\Big);$
      $\Big(InterpreterNewStackFrame[AperiodicEventHandler/class?, APEHinit/methodID?, poppedArgs/methodArgs?]\Big);$
      $Poll\ ;\ AperiodicEventHandler\_APEHinit)\ ;\ pc := 6\ ;\ Poll\ ;\ HandleReturnEPC$
    $\ldots$
    $[]\ pc = 7 \longrightarrow HandleNewEPC(27)\ ;\ pc := 8\ ;\ Poll\ ;\ HandleDupEPC\ ;\ pc := 9\ ;\ Poll\ ;\ HandleAconst\_nullEPC;$
      $pc := 10\ ;\ Poll\ ;\ (\mathbf{var}\ poppedArgs : Word \bullet \Big(\exists\ argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\Big);$
      $\Big(InterpreterNewStackFrame[ConsoleConnection/class?, CCinit/methodID?]\Big)\ ;\ Poll;$
      $ConsoleConnection\_CCinit)\ ;\ pc := 11\ ;\ Poll\ ;\ HandleAstoreEPC(1)\ ;\ pc := 12\ ;\ Poll;$
      $HandleAloadEPC(1)\ ;\ pc := 13\ ;\ Poll\ ;\ (\mathbf{var}\ poppedArgs : Word \bullet$
      $\Big(\exists\ argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\Big)\ ;\ getClassIDOf!head\ poppedArgs?cid \longrightarrow$
      $\mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
        $\Big(InterpreterNewStackFrame[ConsoleConnection/class?, openInputStream/methodID?]\Big);$
        $Poll\ ;\ ConsoleConnection\_openInputStream$
      $\mathbf{fi})\ ;\ pc := 14\ ;\ Poll\ ;\ HandleAstoreEPC(2)\ ;\ pc := 15\ ;\ Poll\ ;\ HandleAloadEPC(1)\ ;\ pc := 16\ ;\ Poll;$
      $(\mathbf{var}\ poppedArgs : Word \bullet \Big(\exists\ argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\Big);$
      $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
        $\Big(InterpreterNewStackFrame[ConsoleConnection/class?, openOutputStream/methodID?]\Big)\ ;\ Poll;$
        $ConsoleConnection\_openOutputStream$
      $\mathbf{fi})\ ;\ pc := 17\ ;\ Poll\ ;\ HandleAstoreEPC(3)\ ;\ pc := 18\ ;\ Poll\ ;\ HandleIconstEPC(0)\ ;\ pc := 19\ ;\ Poll;$
      $HandleAstoreEPC(4)\ ;\ pc := 20\ ;\ Poll\ ;\ pc := 39$
    $[]\ pc = 21 \longrightarrow HandleAloadEPC(2)\ ;\ pc := 22\ ;\ Poll\ ;\ (\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet$
      $\Big(\exists\ argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\Big)\ ;\ getClassIDOf!(head\ poppedArgs)?cid \longrightarrow$
      $\mathbf{if}\ cid = ConsoleInputClassID \longrightarrow$
        $\Big(InterpreterNewStackFrame[ConsoleInput/class?, read/methodID?, poppedArgs/methodArgs?]\Big);$
        $Poll\ ;\ ConsoleInput\_read$
      $\mathbf{fi})\ ;\ pc := 23\ ;\ Poll\ ;\ (\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \Big(\exists\ argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\Big);$
      $\Big(InterpreterNewStackFrame[TPK/class?, f/methodID?, poppedArgs/methodArgs?]\Big)\ ;\ Poll\ ;\ TPK\_f)\ ;\ pc := 24;$
      $Poll\ ;\ HandleAstoreEPC(5)\ ;\ pc := 25\ ;\ Poll\ ;\ HandleAloadEPC(5)\ ;\ pc := 26\ ;\ HandleIconstEPC(400);$
      $pc := 27\ ;\ Poll\ ;\ \mathbf{var}\ value1, value2 : Word \bullet \Big(InterpreterPopEPC[value2!/value!]\Big);$
      $\Big(InterpreterPopEPC[value1!/value!]\Big)\ ;\ pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ 32\ \mathbf{else}\ 28$
    $[]\ pc = 28 \longrightarrow HandleAloadEPC(3)\ ;\ pc := 29\ ;\ Poll\ ;\ HandleIconstEPC(0)\ ;\ pc := 30\ ;\ Poll;$
      $(\mathbf{var}\ poppedArgs : Word \bullet \Big(\exists\ argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\Big);$
      $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
        $\Big(InterpreterNewStackFrame[ConsoleOutput/class?, write/methodID?]\Big)\ ;\ Poll\ ;\ ConsoleOutput\_write$
      $\mathbf{fi})\ ;\ pc := 31\ ;\ Poll\ ;\ pc := 35$
    $[]\ pc = 32 \longrightarrow HandleAloadEPC(3)\ ;\ pc := 33\ ;\ Poll\ ;\ HandleAloadEPC(5)\ ;\ pc := 34\ ;\ Poll;$
      $(\mathbf{var}\ poppedArgs : Word \bullet \Big(\exists\ argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\Big);$
      $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
        $\Big(InterpreterNewStackFrame[ConsoleOutput/class?, write/methodID?]\Big)\ ;\ Poll\ ;\ ConsoleOutput\_write$
      $\mathbf{fi})\ ;\ pc := 35$
    $[]\ pc = 35 \longrightarrow HandleAloadEPC(4)\ ;\ pc := 36\ ;\ Poll\ ;\ HandleIconstEPC(1)\ ;\ pc := 37\ ;\ Poll;$
      $HandleIaddEPC\ ;\ pc := 38\ ;\ Poll\ ;\ HandleAstoreEPC(4)\ ;\ pc := 39$
    $[]\ pc = 39 \longrightarrow HandleAloadEPC(4)\ ;\ pc := 40\ ;\ Poll\ ;\ HandleIconstEPC(10)\ ;\ pc := 41\ ;\ Poll;$
      $\mathbf{var}\ value1, value2 : Word \bullet \Big(InterpreterPopEPC[value2!/value!]\Big);$
      $\Big(InterpreterPopEPC[value1!/value!]\Big)\ ;\ pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ 21\ \mathbf{else}\ 42$
    $[]\ pc = 42 \longrightarrow HandleReturnEPC$
    $\ldots$
    $[]\ pc = 43 \longrightarrow TPK\_f$
    $\ldots$
    $\mathbf{fi}\ ;\ Poll\ ;\ Running$
  $\mathbf{fi}$
```

Figure 5.11: The *Running* action after method call resolution

```
$Running \mathrel{\widehat=}$
  $\mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip}$
  $[\!]\ frameStack \neq \varnothing \longrightarrow$
    $\mathbf{if}\ pc = 0 \longrightarrow TPK\_APEHinit$
    $\ldots$
    $[\!]\ pc = 7 \longrightarrow TPK\_handleAsyncEvent$
    $\ldots$
    $[\!]\ pc = 43 \longrightarrow TPK\_f$
    $\ldots$
    $\mathbf{fi}\ ;\ Poll\ ;\ Running$
  $\mathbf{fi}$
```

Figure 5.12: The *Running* action after all the methods are separated

```
$ExecuteMethod \mathrel{\widehat=}$
    $\mathbf{val}\ classID : ClassID;\ \mathbf{val}\ methodID : MethodID;\ \mathbf{val}\ methodArgs : \operatorname{seq} Word \bullet$
    $\mathbf{if}(classID, methodID) = (TPKClassID, APEHinit) \longrightarrow$
        $InterpreterNewStackFrame[TPK / class?]\ ;\ TPK\_APEHinit$
    $[\!]\ (classID, methodID) = (TPKClassID, handleAsyncEvent) \longrightarrow$
        $InterpreterNewStackFrame[TPK / class?]\ ;\ TPK\_handleAsyncEvent$
    $[\!]\ (classID, methodID) = (TPKClassID, f) \longrightarrow$
        $InterpreterNewStackFrame[TPK / class?]\ ;\ TPK\_f$
    $\ldots$
    $\mathbf{fi}$
```

$$
\begin{array}{l}
MainThread \mathrel{\widehat=} \\
\quad setStack?t : (t = thread)?stack \longrightarrow frameStackID := Initialised\ stack\ ;\ \mu X \bullet \\
\quad \left(\begin{array}{l}
executeMethod?t : (t = thread)?c?m?a \longrightarrow ExecuteMethod(c, m, a)\ ;\ Poll\ ;\ X \\
\square \\
CEEswitchThread?from?to : (from = thread) \longrightarrow Blocked\ ;\ X
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
Started \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
executeMethod?t : (t = thread)?c?m?a \longrightarrow ExecuteMethod(c, m, a)\ ;\ Poll; \\
\qquad \left(\begin{array}{l}
continue?t : (t = thread) \longrightarrow Started \\
\square \\
endThread?t : (t = thread) \longrightarrow \mathbf{Skip}
\end{array}\right) \\
\square \\
CEEswitchThread?from?to : (from = thread) \longrightarrow Blocked\ ;\ Started \\
\square \\
endThread?t : (t = thread) \longrightarrow \mathbf{Skip}
\end{array}\right); \\
\quad removeThreadMemory!thread \longrightarrow SendThread \longrightarrow Sreport?r : (r = Sokay) \\
\quad \longrightarrow CEEswitchThread?from?to : (from = thread) \longrightarrow NotStarted
\end{array}
$$

Figure 5.13: The *ExecuteMethod*, *MainThread*, and *Started* actions after main action refinement

```
$TPK\_handleAsyncEvent \mathrel{\widehat=}$
  $HandleNewEPC(27) \mathbin{;}\ Poll \mathbin{;}\ HandleDupEPC \mathbin{;}\ Poll \mathbin{;}\ HandleAconst\_nullEPC \mathbin{;}$
  $Poll \mathbin{;}\ (\mathbf{var}\ poppedArgs : Word \bullet \left(\exists\, argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}$
  $\left(InterpreterNewStackFrame[ConsoleConnection/class?, CCinit/methodID?]\right) \mathbin{;}\ Poll \mathbin{;}$
  $ConsoleConnection\_CCinit) \mathbin{;}\ Poll \mathbin{;}\ HandleAstoreEPC(1) \mathbin{;}\ Poll \mathbin{;}$
  $HandleAloadEPC(1) \mathbin{;}\ Poll \mathbin{;}\ (\mathbf{var}\ poppedArgs : Word \bullet$
  $\left(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}\ getClassIDOf!head\ poppedArgs?cid \longrightarrow$
  $\mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
    $\left(InterpreterNewStackFrame[ConsoleConnection/class?, openInputStream/methodID?]\right) \mathbin{;}$
    $Poll \mathbin{;}\ ConsoleConnection\_openInputStream$
  $\mathbf{fi}) \mathbin{;}\ Poll \mathbin{;}\ HandleAstoreEPC(2) \mathbin{;}\ Poll \mathbin{;}\ HandleAloadEPC(1) \mathbin{;}\ Poll \mathbin{;}$
  $(\mathbf{var}\ poppedArgs : Word \bullet \left(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}$
  $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
    $\left(InterpreterNewStackFrame[ConsoleConnection/class?, openOutputStream/methodID?]\right) \mathbin{;}\ Poll \mathbin{;}$
    $ConsoleConnection\_openOutputStream$
  $\mathbf{fi}) \mathbin{;}\ Poll \mathbin{;}\ HandleAstoreEPC(3) \mathbin{;}\ Poll \mathbin{;}\ HandleIconstEPC(0) \mathbin{;}\ Poll \mathbin{;}$
  $HandleAstoreEPC(4) \mathbin{;}\ Poll \mathbin{;}\ Poll \mathbin{;}\ (\mu\, Y \bullet$
    $HandleAloadEPC(4) \mathbin{;}\ Poll \mathbin{;}\ HandleIconstEPC(10) \mathbin{;}\ Poll \mathbin{;}$
    $(\mathbf{var}\ value1, value2 : Word \bullet \left(InterpreterPopEPC[value2!/value!]\right) \mathbin{;}\ \left(InterpreterPopEPC[value1!/value!]\right)$
    $\mathbf{if}\ value1 \leq value2 \longrightarrow Poll \mathbin{;}\ HandleAloadEPC(2) \mathbin{;}\ Poll \mathbin{;}$
      $(\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \left(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}$
      $getClassIDOf!(head\ poppedArgs)?cid \longrightarrow \mathbf{if}\ cid = ConsoleInputClassID \longrightarrow$
        $\big(InterpreterNewStackFrame[$
          $ConsoleInput/class?, read/methodID?, poppedArgs/methodArgs?]\big) \mathbin{;}$
        $Poll \mathbin{;}\ ConsoleInput\_read$
      $\mathbf{fi}) \mathbin{;}\ Poll \mathbin{;}\ (\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \left(\exists\, argsToPop? == 1 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}$
      $\left(InterpreterNewStackFrame[TPK/class?, f/methodID?, poppedArgs/methodArgs?]\right) \mathbin{;}\ Poll \mathbin{;}\ TPK\_f) \mathbin{;}$
      $Poll \mathbin{;}\ HandleAstoreEPC(5) \mathbin{;}\ Poll \mathbin{;}\ HandleAloadEPC(5) \mathbin{;}\ HandleIconstEPC(400) \mathbin{;}\ Poll \mathbin{;}$
      $(\mathbf{var}\ value1, value2 : Word \bullet \left(InterpreterPopEPC[value2!/value!]\right) \mathbin{;}\ \left(InterpreterPopEPC[value1!/value!]\right) \mathbin{;}$
      $\mathbf{if}\ value1 \leq value2 \longrightarrow HandleAloadEPC(3) \mathbin{;}\ Poll \mathbin{;}\ HandleAloadEPC(5) \mathbin{;}$
        $Poll \mathbin{;}\ (\mathbf{var}\ poppedArgs : Word \bullet \left(\exists\, argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}$
        $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
          $\left(InterpreterNewStackFrame[ConsoleOutput/class?, write/methodID?]\right) \mathbin{;}$
          $Poll \mathbin{;}\ ConsoleOutput\_write$
        $\mathbf{fi})$
      $[]\ value1 > value2 \longrightarrow HandleAloadEPC(3) \mathbin{;}\ Poll \mathbin{;}\ HandleIconstEPC(0) \mathbin{;}$
        $Poll \mathbin{;}\ (\mathbf{var}\ poppedArgs : Word \bullet \left(\exists\, argsToPop? == 2 \bullet InterpreterStackFrameInvokeEPC\right) \mathbin{;}$
        $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
          $\left(InterpreterNewStackFrame[ConsoleOutput/class?, write/methodID?]\right) \mathbin{;}$
          $Poll \mathbin{;}\ ConsoleOutput\_write$
        $\mathbf{fi}) \mathbin{;}\ Poll$
      $\mathbf{fi}) \mathbin{;}\ Poll \mathbin{;}\ HandleAloadEPC(4) \mathbin{;}\ Poll \mathbin{;}\ HandleIconstEPC(1) \mathbin{;}\ Poll \mathbin{;}$
      $HandleIaddEPC \mathbin{;}\ Poll \mathbin{;}\ HandleAstoreEPC(4) \mathbin{;}\ Poll \mathbin{;}\ Y$
    $[]\ value1 > value2 \longrightarrow \mathbf{Skip}$
    $\mathbf{fi})) \mathbin{;}\ Poll \mathbin{;}\ HandleReturnEPC$
```

Figure 5.14: The *TPK_handleAsyncEvent* action after *pc* has been eliminated from the state

conditionals have been introduced, all the methods have been separated out, and all method calls have been resolved, is shown in Figure 5.12. At this point, the actions at $pc = 0$ have been separated into a *TPK_APEHinit* action, and the actions at $pc = 7$ have been separated into a *TPK_handleAsyncEvent* action. We omit the definitions of these actions, since they are just the contents of the $pc = 0$ and $pc = 7$ branches in Figure 5.9. The *pc* values for the other branches are now redundant, since their instructions have been folded into the method actions.

The next step is then to eliminate the redundant paths and remove the dependency on *pc* to select the method action. This occurs at line 8 of Algorithm 1, in which the *Started* and *MainThread* actions are refined to replace the *Running* action with an *ExecuteMethod* action that contains a choice of method action based on the method and class identifiers of the method. Figure 5.13 shows the *ExecuteMethod* action corresponding to our example, and the refined *MainThread* and *Started* actions that reference it. We describe this refinement in more detail in Section 5.3.6, where we define the RefineMainActions procedure.

When all of the previous steps are completed, reliance on *pc* to determine control flow has been completely removed. The *pc* state component can then be removed in a simple data refinement that also removes all the assignments to *pc*, resulting in the *TPK_handleAsyncEvent* action shown in Figure 5.14. The data refinement to remove *pc* is applied at the end of the algorithm, on line 9, and is described in more detail in Section 5.3.7, where we define the RemovePCFromState procedure.

The remaining instruction handling actions then only affect the stack, the removal of which is the concern of the next stage of the compilation strategy.

We now proceed to describe each of the steps of Algorithm 1 in more detail.

5.3.2 Expand Bytecode

Before the control flow can be introduced, the bytecode instructions provided in the *bc* parameter to *Thr* must be expanded to allow consideration of their semantics. This is performed as specified in Algorithm 2, which defines the ExpandBytecode procedure. It begins on line 1

Algorithm 2 ExpandBytecode
```
 1: apply Rule [pc-expansion](bc)
 2: for (handleActionName, handleActionBody) ← Handle*EPCActions do
 3:     apply Law [action-intro](handleActionName, handleActionBody)
 4: end for
 5: for i ← dom bc do
 6:     apply Rule [HandleInstruction-refinement](bc, i)
 7:     try
 8:         apply Rule [CheckSynchronizedReturn-sync-refinement](i)
 9:         apply Rule [CheckSynchronizedReturn-nonsync-refinement](i)
10:     end try
11: end for
```

by applying Rule [*pc*-expansion], shown in Figure 5.15. It introduces a choice over all the possible values of *pc* in the domain of *bc* at the *HandleInstruction* action in *Running*. This does not affect the behaviour of *HandleInstruction*, because it behaves as **Chaos** when *pc* is outside the domain of *bc*. We write *HandleInstruction* with a *bc* subscript to indicate that it makes use of *bc*, which is a parameter of the *Thr* process in which *HandleInstruction* occurs. The proof of this rule and others can be found in Appendix G of the extended version of this thesis [13].

After applying Rule [*pc*-expansion], we operate on the occurrence of *HandleInstruction* at each branch of the conditional at line 5. We apply Rule [*HandleInstruction*-refinement], shown in Figure 5.16, on line 6 to refine each occurrence to a more specific form that is easier to operate on during the rest of the strategy. These new actions are determined from the bytecode

Rule [*pc*-expansion]**.** Given $bc : ProgramAddress \pfun Bytecode$,

$$HandleInstruction_{bc} = \mathbf{if}\ \Box_{i \in \operatorname{dom} bc}\ pc = i \longrightarrow HandleInstruction_{bc}\ \mathbf{fi}$$

Figure 5.15: Rule [*pc*-expansion]

instruction in *bc* at each *pc* value by applying a syntactic function *handleAction*, which is defined by Table 5.1.

Rule [*HandleInstruction*-refinement]**.** Given $i : ProgramAddress$, if $i \in \operatorname{dom} bc$ then,

$$\begin{array}{l} \mathbf{if}\ \cdots \\ \Box\ pc = i \longrightarrow HandleInstruction_{bc} \\ \ldots \\ \mathbf{fi} \end{array} \quad \sqsubseteq_A \quad \begin{array}{l} \mathbf{if}\ \cdots \\ \Box\ pc = i \longrightarrow handleAction(bc\ i) \\ \ldots \\ \mathbf{fi} \end{array}$$

where *handleAction* is a syntactic function defined by Table 5.1.

Figure 5.16: Rule [*HandleInstruction*-refinement]

The actions generated by *handleAction* use new actions for handling the individual bytecode instructions. These are similar to the actions used to define *HandleInstruction* (e.g. *HandleDup*, *HandleAload* etc.), which we refer to as *Handle∗* actions. We name the new actions used by *handleAction* by appending *EPC* to the names of the *Handle∗* actions they are based on, and we refer to them as *Handle∗EPC* actions. The *Handle∗EPC* actions are introduced in the for loop starting on line 2 of Algorithm 2, before the application of Rule [*HandleInstruction*-refinement], by application of Law [action-intro], which introduces unused actions to processes. These are actions of a fixed form, described below, so we can introduce them directly.

In addition to the *Handle∗EPC* actions, the actions output from *handleAction* also include *pc* updates extracted from the *Handle∗* actions. The output from *handleAction* for the *goto* and *if_icmple* instructions consists solely of a *pc* update with no *Handle∗EPC* actions, since updating the value of pc is the main effect of those instructions.

The differences between the *Handle∗EPC* actions and the *Handle∗* actions on which they are based are explained using the *HandleAstore* action as an example. We recall that it is defined as shown below.

$$\begin{array}{l} HandleAstore \mathrel{\widehat=} (pc \in \operatorname{dom} bc \land bc\,pc \in \operatorname{ran} astore) \& \\ \quad \mathbf{var}\ variableIndex : \mathbb{N} \bullet variableIndex := (astore^{\sim})\,(bc\,pc)\ ;\ \big(InterpreterAstore\big) \end{array}$$

Its corresponding *Handle∗EPC* action, *HandleAstoreEPC*, is shown below.

$$HandleAstoreEPC \mathrel{\widehat=} \mathbf{val}\ variableIndex : \mathbb{N} \bullet \big(InterpreterAstoreEPC\big)$$

The first difference of *HandleAstoreEPC* from *HandleAstore* is that it is not guarded by the condition on the value of *bc* at the current *pc* value. The choice that such guards mediate is collapsed by Rule [*HandleInstruction*-refinement], since the value of *bc* at a given *pc* value is determined by the supplied *bc* parameter of *Thr*.

The second difference is that the parameters of the bytecode instructions are transferred to become parameters of the *Handle∗EPC* actions, so *HandleAstoreEPC* has a *variableIndex*

Bytecode ($bc\ i$)	Action ($handleAction(bc\ i)$)
$aconst\_null$	$HandleAconst\_nullEPC\ ;\ pc := i + 1$
dup	$HandleDupEPC\ ;\ pc := i + 1$
$aload\ lvi$	$HandleAloadEPC(lvi)\ ;\ pc := i + 1$
$astore\ lvi$	$HandleAstoreEPC(lvi)\ ;\ pc := i + 1$
$iadd$	$HandleIaddEPC\ ;\ pc := i + 1$
$iconst\ n$	$HandleIconstEPC(n)\ ;\ pc := i + 1$
$ineg$	$HandleInegEPC\ ;\ pc := i + 1$
$goto\ ofst$	$pc := i + ofst$
$if\_icmple\ ofst$	$\mathbf{var}\ value1, value2 : Word \bullet$ $\big(InterpreterPopEPC[value2!/value!]\big);$ $\big(InterpreterPopEPC[value1!/value!]\big);$ $pc := \mathbf{if}\ value1 \leq value2\ \mathbf{then}\ i + ofst\ \mathbf{else}\ i + 1$
$areturn$	$CheckSynchronizedReturn\ ;\ HandleAreturnEPC$
$return$	$CheckSynchronizedReturn\ ;\ HandleReturnEPC$
$new\ cpi$	$HandleNewEPC(cpi)\ ;\ pc := i + 1$
$getfield\ cpi$	$HandleGetfieldEPC(cpi)\ ;\ pc := i + 1$
$putfield\ cpi$	$HandlePutfieldEPC(cpi)\ ;\ pc := i + 1$
$getstatic\ cpi$	$HandleGetstaticEPC(cpi)\ ;\ pc := i + 1$
$putstatic\ cpi$	$HandlePutstaticEPC(cpi)\ ;\ pc := i + 1$
$invokevirtual\ cpi$	$\{pc = i\}\ ;\ HandleInvokevirtualEPC(cpi)$
$invokespecial\ cpi$	$\{pc = i\}\ ;\ HandleInvokespecialEPC(cpi)$
$invokestatic\ cpi$	$\{pc = i\}\ ;\ HandleInvokestaticEPC(cpi)$

Table 5.1: The syntactic function *handleAction*

parameter. This corresponds to the *variableIndex* variable in *HandleAstore*, which is used to store the value extracted from the *astore* instruction. This transformation is, of course, not performed for instructions that do not take parameters. This transformation is standard in the context of a call to a parametrised action.

Finally, the schema *InterpreterAstore* is replaced with a schema *InterpreterAstoreEPC*, which does not affect *pc*, since Rule [*HandleInstruction*-refinement] extracts the updates to *pc* from the *Handle*∗ actions. The *pc* updates are not removed in the case of the actions for handling method invocation and return, where the *pc* updates are closely connected to the operations on the stack and require special handling. Instead, an assumption on the value of *pc* is introduced for the method invocation handling actions, since the *pc* information is used in setting the return address. We discuss how we operate on the method invocation and return handling actions in Section 5.3.5.

We also note that the *CheckSynchronizedReturn* action is moved outside the *Handle*∗*EPC* actions handling return instructions. This is so that this action can be removed, since we have sufficient information to determine whether the method is synchronized or not. This is handled on lines 8 and 9 of Algorithm 2, by the application of Rule [*CheckSynchronizedReturn*-sync-refinement] and Rule [*CheckSynchronizedReturn*-nonsync-refinement]. These rules are applied in a try block, beginning on line 7, which tries to apply each rule in turn, stopping when one succeeds.

Rule [*CheckSynchronizedReturn*-sync-refinement] matches a branch of the choice in *Running* corresponding to a given *pc* value, i, which begins with a *CheckSynchronizedReturn* action.

This rule is applied whenever the unique class, c, and method, m, in which i occurs are such that m is synchronized and not static in c. The rule refines *CheckSynchronizedReturn* to a communication with the *Launcher* on the *releaseLock* channel, instructing it to release the lock on the `this` object. Rule [*CheckSynchronizedReturn*-nonsync-refinement] is similar, but applies when m is static or not synchronized in c, and eliminates *CheckSynchronizedReturn*.

Rule [*CheckSynchronizedReturn*-sync-refinement]**.** Given $i : ProgramAddress$,

$$\begin{array}{l} \mathbf{if} \cdots \\ \;[\!]\; pc = i \longrightarrow \\ \quad CheckSynchronizedReturn \;;\; A \\ \cdots \\ \mathbf{fi} \end{array} \sqsubseteq_A \begin{array}{l} \mathbf{if} \cdots \\ \;[\!]\; pc = i \longrightarrow \\ \quad releaseLock!((last\, frameStack).localVariables\; 1) \\ \quad \longrightarrow releaseLockRet \longrightarrow \mathbf{Skip} \;;\; A \\ \cdots \\ \mathbf{fi} \end{array}$$

provided

$$\begin{array}{l} \exists\, c : Class;\; m : MethodID \mid \\ \quad c \in \operatorname{ran} cs \land m \in \operatorname{dom} c.methodEntry \bullet \\ \quad i \in c.methodEntry\; m \mathrel{..} c.methodEnd\; m \land \\ \quad m \in c.synchronizedMethods \land m \notin c.staticMethods \end{array}$$

Figure 5.17: Rule [*CheckSynchronizedReturn*-sync-refinement]

At the end of Algorithm 2, our example has the form shown earlier in Figure 5.7. After the bytecode semantics is expanded in the *Running* action by this step, the control flow that corresponds to each *pc* update can be introduced. This is discussed in the next section.

5.3.3 Introduce Sequential Composition

Algorithm 3 INTRODUCESEQUENTIALCOMPOSITION

```
1: $cfg \leftarrow$ MAKECONTROLFLOWGRAPH
2: for $node \leftarrow$ NODES($cfg$) do
3:     while HASSIMPLESEQUENCE($node$) do
4:         apply Rule [sequence-intro]($node$)
5:     end while
6: end for
```

The simplest control flows to introduce are those of instructions where execution continues at the next program counter value. These control flows are introduced as shown in Algorithm 3, which defines the INTRODUCESEQUENTIALCOMPOSITION procedure. The algorithm constructs a control flow graph for each method in the program, as specified on line 1. Since the introduction of sequential composition does not depend on the relationships between methods, the control flow graph is constructed as a disconnected graph containing the control flow of each method in the program. The nodes in this graph correspond to the branches in the choice over *pc* values introduced in the previous section.

We construct the control flow graph by starting at the entry point for each method and following the *pc* update at the end of each node, introducing an edge in the process. For method call

$$\begin{tikzcd}
0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \\
7 \arrow[r] & 8 \arrow[r] & 9 \arrow[r] & 10 \arrow[r] & 11 \arrow[r] & 12 \arrow[r] & 13 \arrow[r] & 14 \arrow[r] & 15 \arrow[r] & 16 \arrow[r] & 17 \arrow[r] & 18 \arrow[r] & 19 \arrow[r] & 20 \arrow[ddrr] \\
& & & & & & & 28 \arrow[r] & 29 \arrow[r] & 30 \arrow[r] & 31 \arrow[dr] \\
21 \arrow[r] & 22 \arrow[r] & 23 \arrow[r] & 24 \arrow[r] & 25 \arrow[r] & 26 \arrow[r] & 27 \arrow[ur] \arrow[dr] & & & & & 35 \arrow[r] & 36 \arrow[r] & 37 \arrow[r] & 38 \arrow[r] & 39 \arrow[r] & 40 \arrow[r] & 41 \arrow[r] \arrow[lllllllllllllllll, bend left] & 42 \\
& & & & & & & 32 \arrow[r] & 33 \arrow[r] & 34 \arrow[urr] \\
43 \arrow[r] & 44 \arrow[r] & 45 \arrow[r] & 46 \arrow[r] & 47 \arrow[r] & 48 \arrow[r] & 49 \arrow[r] & 50
\end{tikzcd}$$

Figure 5.18: Control flow graph for our example program

instructions, we introduce an edge to the node for the next *pc* value, as if the instruction were replaced with $pc := pc + 1$. This is consistent with how method calls are handled later in the strategy, since execution resumes at the next instruction after the called method returns. We do not add an edge from a return instruction, since no further instructions are executed in the method after a return instruction. Construction of the control graph for a method terminates when there are no further edges to add. Since there are only finitely many instructions in a method, edges for all the reachable nodes are eventually added.

The control flow graph for our example is shown in Figure 5.18. We label the nodes of the graph with the *pc* values of the instructions at the nodes. Due to our assumptions about the source bytecode, the subgraph corresponding to each method's control flow is a structured graph as defined in Section 5.2.

After the control flow graph is constructed, we consider each node in turn, as specified by the for loop starting on line 2. As mentioned earlier, we require a node to have only a single outgoing edge and its target to have only a single incoming edge in order for it to be considered for the introduction of sequential composition. The reason for this is that nodes with two outgoing edges are points at which conditionals should be introduced. Such nodes in our example are the nodes for *pc* values 27 and 41, which represent the start of conditionals. Likewise, nodes with multiple incoming edges represent points at which a more complex control flows occur. For our example, such nodes include 39, which is the start of a loop, and 35, which is the end of a conditional. These prevent introduction of sequential composition for the *pc* values 20, 31, 34, and 38, since the targets of those nodes are nodes 35 and 39.

The procedure HasSimpleSequence(*node*) checks this requirement for introducing sequential composition. It returns a true value if *node* has only a single outgoing edge and its target has only a single incoming edge, and otherwise returns a false value. This check is performed on line 3, where it defines the condition of a while loop.

For a node that meets the above requirement and is not a method call, we can introduce sequential composition at that node by applying Rule [sequence-intro] (Figure 5.19), on line 4 of the algorithm. This rule works by unrolling the loop in *Running* to sequence an instruction at *pc* value i with the instruction that is executed after it, inserting *Poll* inbetween. It is required that the *pc* value of the node's target, j, not be the same as i, since that would introduce a loop, rather than a sequential composition. Also, the sequence of instructions at the node, A, must not affect the non-emptiness of the *frameStack* to ensure that the choice at the start of the main loop in *Running* can be resolved.

Rule [sequence-intro]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$ and

$$\begin{array}{l}
\{\mathit{frameStack} \neq \varnothing\} \; ; \; A \\
= \\
\{\mathit{frameStack} \neq \varnothing\} \; ; \; A \; ; \; \{\mathit{frameStack} \neq \varnothing\}
\end{array}$$

then,

$$\begin{array}{lcl}
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!] \; \mathit{frameStack} \neq \varnothing \longrightarrow \\
\qquad \mathbf{if} \cdots \\
\qquad [\!] \; pc = i \longrightarrow A \; ; \; pc := j \\
\qquad \ldots \\
\qquad [\!] \; pc = j \longrightarrow B \\
\qquad \ldots \\
\qquad \mathbf{fi} \; ; \; \mathit{Poll} \; ; \; X \\
\quad \mathbf{fi}
\end{array}
& \sqsubseteq_A &
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!] \; \mathit{frameStack} \neq \varnothing \longrightarrow \\
\qquad \mathbf{if} \cdots \\
\qquad [\!] \; pc = i \longrightarrow \\
\qquad\quad A \; ; \; pc := j \; ; \; \mathit{Poll} \; ; \; B \\
\qquad \ldots \\
\qquad [\!] \; pc = j \longrightarrow B \\
\qquad \ldots \\
\qquad \mathbf{fi} \; ; \; \mathit{Poll} \; ; \; X \\
\quad \mathbf{fi}
\end{array}
\end{array}$$

Figure 5.19: Rule [sequence-intro]

Since Rule [sequence-intro] pulls two nodes together, we can continue to introduce sequential composition at a node after the first application of Rule [sequence-intro], until that node no longer satisfies the conditions for introducing sequential composition. This is specified by the while loop starting at line 3 of the algorithm. The control flow graph is updated as Rule [sequence-intro] is applied, to take into account the merging of nodes. Since there are finitely many nodes, the merging of nodes eventually results in a graph in which no further sequential compositions can be introduced and so the loop terminates.

The resulting control flow graph after introduction of sequential composition has been performed at every point is shown in Figure 5.20. We note that this graph is still a union of structured graphs since merging sequentially composed nodes does not affect whether a graph is structured. This is due to the fact that sequential composition is one of the constructs used to define structured control flow graphs (Figure 5.1a), and merging the nodes may be seen as performing the reverse of node replacement for it.

The only remaining nodes in this graph are those where the sequence of instructions ends with a method call or return, or which represent a more complex control flow. In particular, the instructions for the `f()` method of `TPK`, which begin at $pc = 43$, have been completely sequenced together into a single node. The code that corresponds to this control flow graph is that shown earlier in Figure 5.8.

5.3.4 Introduce Loops and Conditionals

After sequential composition has been introduced for all methods, we must consider each method separately to handle method calls. This means the strategy must loop, introducing loops and

$$\begin{tikzcd}
0 \arrow[r] & 6 & & & & & & \\
7 \arrow[r] & 11 \arrow[r] & 14 \arrow[r] & 17 \arrow[ddrrr] & & & & \\
& & & 28 \arrow[r] & 31 \arrow[dr] & & & \\
21 \arrow[r] & 23 \arrow[r] & 24 \arrow[ur] \arrow[dr] & & & 35 \arrow[r] & 39 \arrow[r] \arrow[llllll, bend left] & 42 \\
& & & 32 \arrow[urr] & & & & \\
43 & & & & & & &
\end{tikzcd}$$

Figure 5.20: Control flow graph for our example after sequential composition introduction

conditionals to those methods that have no unresolved method calls and resolving calls of methods that are then complete, until every method is complete and has been separated into its own action.

Introducing loops and conditionals is performed as described by Algorithm 4. This considers each method individually, collecting the list of method entries from the *cs* information on line 1, and iterating over them in the for loop on line 2. The program address that forms the entry point of the method under consideration in a given iteration of the loop is referred to as *methodEntry*. The condition on line 3 ensures that only those methods where all method calls have already been resolved undergo loop and conditional introduction. Since we do not allow recursion, there is always at least one method that does not depend on another method in the program. It may be the case that a method depends only on special methods, in which case this stage has no effect on that method until the special method calls have been resolved. Special method calls can always be resolved as they do not depend on other methods in the program.

The HasNoUresolvedCalls(*methodEntry*) procedure, used in the condition on line 3, checks that no node in the control flow graph beginning at *methodEntry* ends in a method call, as a way of determining whether the method has unresolved calls. Since method resolution sequences a method call with the instructions following it, a method call with nothing following it is a call that has not yet been resolved.

For each method that undergoes loop and conditional introduction, we consider again its control-flow graph to ensure the loops and conditionals are introduced in the correct order to properly form their bodies. This involves constructing a control-flow graph for the method, at line 4. The control-flow graph for a method beginning at *methodEntry* is created by the procedure MakeMethodControlFlowGraph(*methodEntry*). This is similar to the MakeControlFlowGraph procedure used in the previous section, but it just constructs the graph for a single method, starting at its *methodEntry*.

The graph for the `handleAsyncEvent()` method in our example (beginning at $pc = 7$, its entry point) is shown in Figure 5.21, alongside the *Circus* code obtained at the beginning of this stage for the method. The edge that forms a loop from $pc = 35$ to $pc = 39$ is shown as a dashed line since looping edges are ignored at certain points in this part of the strategy.

The control-flow graph of each method is structured since the transformations of the graph up to this point consist solely of collapsing sequential compositions, which, as explained in the previous section, does not cause a structured graph to become unstructured. Since we have

Algorithm 4 INTRODUCELOOPSANDCONDITIONALS

```
 1: methodEntries ← METHODENTRIES(cs)
 2: for methodEntry ← methodEntries do
 3:     if HASNOURESOLVEDCALLS(methodEntry) then
 4:         cfg ← MAKEMETHODCONTROLFLOWGRAPH(methodEntry)
 5:         iterationOrder ← REVERSENODES(cfg)
 6:         for node ← iterationOrder do
 7:             apply Rule [if-conditional-intro](node)
 8:             apply Rule [if-else-conditional-intro](node)
 9:             if ISSIMPLECONDITIONAL(node) then
10:                 apply Rule [conditional-intro](node)
11:             end if
12:             apply Rule [while-loop-intro1](node)
13:             apply Rule [while-loop-intro2](node)
14:             apply Rule [do-while-loop-intro](node)
15:             apply Rule [infinite-loop-intro](node)
16:             if HASSIMPLESEQUENCE(node) then
17:                 apply Rule [sequence-intro](node)
18:             end if
19:         end for
20:     end if
21: end for
```

$$\begin{tikzcd}
 & 7 \arrow[d] & & & \\
 & 39 \arrow[dl] \arrow[drr] & & & \\
42 & & & 21 \arrow[dl] \arrow[dr] & \\
 & & 28 \arrow[dr] & & 32 \arrow[dl] \\
 & & & 35 \arrow[uuull, dashed, bend right] &
\end{tikzcd}$$

```
Running ≙
    if frameStack = ∅ ⟶ Skip
    [] frameStack ≠ ∅ ⟶
        if pc = 0 ⟶ · · ·
        [] pc = 7 ⟶ HandleNewEPC(27) ⨾ pc := 8 ⨾ Poll ⨾ · · · ⨾
              pc := 39
        . . .
        [] pc = 21 ⟶ HandleAloadEPC(2) ⨾ pc := 22 ⨾ Poll ⨾ · · · ⨾
              pc := if value1 ≤ value2 then 32 else 28
        . . .
        [] pc = 28 ⟶ HandleAloadEPC(3) ⨾ pc := 29 ⨾ Poll ⨾ · · · ⨾
              pc := 35
        . . .
        [] pc = 32 ⟶ HandleAloadEPC(3) ⨾ pc := 33 ⨾ Poll ⨾ · · · ⨾
              pc := 35
        . . .
        [] pc = 35 ⟶ HandleAloadEPC(4) ⨾ pc := 36 ⨾ Poll ⨾ · · · ⨾
              pc := 39
        . . .
        [] pc = 39 ⟶ HandleAloadEPC(4) ⨾ pc := 36 ⨾ Poll ⨾ · · · ⨾
              pc := if value1 ≤ value2 then 21 else 42
        . . .
        [] pc = 42 ⟶ HandleReturnEPC
        fi ⨾ Poll ⨾ Running
    fi
```

Figure 5.21: Simplified control flow graph and corresponding code for our example program

defined the desired program structure in terms of a small number of standard structures (shown in Figure 5.1), we can identify each of these structures in the graph and introduce them into the program, collapsing the graph in the process.

We iterate over the nodes in the method's control flow graph, identifying the control flow structures at each node. This is specified by the loop beginning on line 6 of Algorithm 4.

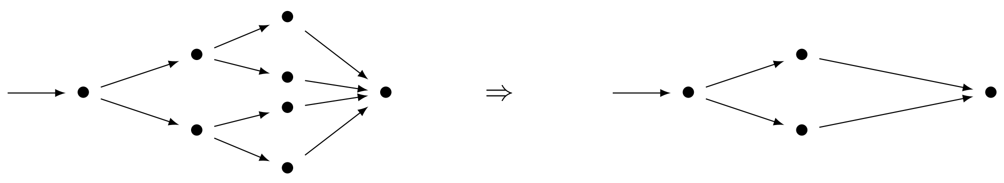
In order to identify a structure, we must first introduce any structures embedded in it. This can be seen by considering a graph such as that shown in Figure 5.3c, where the graph starting at node 1 does not have the form of an `if-else` conditional (Figure 5.1c), although it is constructed from such a graph. The subgraph starting at a, however, does have the form of a `while` loop, so it can be introduced. Once that has been introduced, the graph has the form of an `if-else` conditional. To introduce such embedded structures first, we consider the successors of each node (ignoring loops) before the node itself. This ensures that we consider the internal nodes of a structure first and introduce any structures that may have been inserted at those nodes via internal-node or branch-end replacement.

The iteration order is specified using a procedure ReverseNodes(cfg), which constructs a sequence indicating the order in which the nodes of cfg should be iterated over. The sequence is constructed ensuring that, where a node occurs in the sequence, the successors of that node (ignoring loops) occur earlier. This means that the sequence begins with an end node (ignoring loops). The order, $iterationOrder$, is constructed on line 5 of the algorithm and used for the range of the for loop starting on line 6.

In our example, we may consider the $pc = 42$ and $pc = 35$ nodes first, then $pc = 28$ and $pc = 32$, then $pc = 21$, $pc = 39$, and finally $pc = 7$, as can be seen from the graph in Figure 5.21. Other valid orders may be used in an implementation of the strategy.

For each node, we check each type of structure to see if the control-flow graph starting at that point matches the structure, and introduce the structure if it does. Some of the structures (Figure 5.1b, c, e and f) are followed by further instructions. In these cases, a sequential composition must be introduced with the instructions following the structure.

However, in programs with graphs such as the one shown on the left below, the sequential composition cannot be introduced after the inner conditionals have been introduced. Introducing the inner conditionals yields the graph shown on the right below, which has the form of an `if-else` conditional. This graph would be broken up by the introduction of a sequential composition to the final node, since it is part of the outer conditional. Thus, the introduction of the sequential composition cannot be made part of the rule for introducing the inner conditional. We instead perform sequential compositions for such structures separately, rather than as part of the loop and conditional introduction rules.



The first type of structure we check for are conditionals. There are three conditional structures: `if` conditionals (Figure 5.1b), `if-else` conditionals (Figure 5.1c), and divergent conditionals (Figure 5.1d). We introduce each with a separate rule, specialised to the form of the conditional.

An `if` conditional with no else branch is introduced using Rule [`if`-conditional-intro], shown in Figure 5.22. Such a structure can be recognised from the form of the *Circus* code in the *Running* action, which is that of a node whose sequence of instructions ends with a variable block, ending with an assignment of the form $pc := \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y$, and for which the $pc = y$ node ends in an assignment $pc := x$. Note that the branches cannot be the other way round (i.e. the $pc = x$ branch cannot be the body of the conditional) since the conditional branches come from Java's branching instructions, which branch to the specified address if the condition is true and go to the next instruction if it is false.

Rule [`if`-conditional-intro]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$, $i \neq k$, and

$$\begin{array}{l} \{\mathit{frameStack} \neq \varnothing\} \mathbin{;}\ A \mathbin{;}\ P \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \mathbin{;}\ A \mathbin{;}\ P \mathbin{;}\ \{\mathit{frameStack} \neq \varnothing\} \end{array}$$

then

```
$\mu X \bullet$
    $\mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip}$
    $[]\ \mathit{frameStack} \neq \varnothing \longrightarrow$
        $\mathbf{if} \cdots$
        $[]\ pc = i \longrightarrow A;$
            $(\mathbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            $pc := \mathbf{if}\ b\ \mathbf{then}\ j\ \mathbf{else}\ k)$
        $\cdots$
        $[]\ pc = k \longrightarrow B \mathbin{;}\ pc := j$
        $\cdots$
        $\mathbf{fi} \mathbin{;} \mathit{Poll} \mathbin{;}\ X$
    $\mathbf{fi}$
```

$\sqsubseteq_A$

```
$\mu X \bullet$
    $\mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip}$
    $[]\ \mathit{frameStack} \neq \varnothing \longrightarrow$
        $\mathbf{if} \cdots$
        $[]\ pc = i \longrightarrow A;$
            $(\mathbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            $\mathbf{if}\ b \longrightarrow \mathbf{Skip}$
            $[]\ \neg\ b \longrightarrow pc := k \mathbin{;}\ \mathit{Poll} \mathbin{;}\ B$
            $\mathbf{fi}) \mathbin{;}\ pc := j$
        $\cdots$
        $[]\ pc = k \longrightarrow B \mathbin{;}\ pc := j$
        $\cdots$
        $\mathbf{fi} \mathbin{;} \mathit{Poll} \mathbin{;}\ X$
    $\mathbf{fi}$
```

Figure 5.22: Rule [`if`-conditional-intro]

Rule [`if`-conditional-intro] introduces a conditional for nodes that match the form described above, which in the rule is the $pc = i$ node. The conditional is introduced with the true branch empty (represented by **Skip**) and the false branch containing the instructions in the body of the conditional. The assignment $pc := j$ is moved outside the conditional from both the true and false branches.

As in Rule [sequence-intro], the sequence of actions for the node must not affect the nonemptiness of the *frameStack*. A similar condition is required for all the rules in this section. We also require that the targets of the conditional are different from the node at which the conditional is introduced, since that would introduce a loop, which is not the purpose of this rule. Rule [`if`-conditional-intro] is applied on line 7 of Algorithm 4. Note that, since the structure can be identified from the form of the *Circus* code alone, it is not necessary to guard the application of the rule with a condition on the control-flow graph.

We introduce `if-else` conditionals using Rule [`if-else`-conditional-intro] and divergent conditionals using Rule [conditional-intro]. Since these are similar to Rule [`if`-conditional-intro], we omit them here. They can be found in Appendix A. We apply these rules on lines 8 and 10.

Rule [conditional-intro] introduces a conditional with no restrictions on its form. To ensure it is only applied to nodes that match the form of Figure 5.1d, we guard its application by the condition IsSimpleConditional(*node*) on line 9. The procedure IsSimpleConditional(*node*) checks if the targets of *node* have no outgoing nodes. This is a condition on the control-flow graph that cannot be expressed in the statement of the rule.

After attempting to introduce conditionals, we attempt to introduce loops. There are three types of loop to consider, as shown earlier: `while` loops (Figure 5.1e), `do-while` loops (Figure 5.1f), and infinite loops (Figure 5.1g). A `while` loop has a form similar to that of a conditional, except that one of the branches ends with a jump back to the beginning of the node with the conditional. This structure may be introduced using Rule [`while`-loop-intro1], shown in Figure 5.23. This rule introduces a conditional at a node $pc = i$ with its false branch ending in an assignment of i to pc, and introduces a recursion to the beginning of the $pc = i$ node in that branch of the conditional, representing a loop.

Rule [`while`-loop-intro1]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$,

$$\begin{array}{l} \{\mathit{frameStack} \neq \varnothing\} \mathrel{;} A \mathrel{;} P \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \mathrel{;} A \mathrel{;} P \mathrel{;} \{\mathit{frameStack} \neq \varnothing\} \end{array}$$

and

$$\begin{array}{l} \{\mathit{frameStack} \neq \varnothing\} \mathrel{;} C \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \mathrel{;} C \mathrel{;} \{\mathit{frameStack} \neq \varnothing\} \end{array}$$

then

$$\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!]\, \mathit{frameStack} \neq \varnothing \longrightarrow \\
\qquad \mathbf{if} \cdots \\
\qquad [\!]\, pc = i \longrightarrow A; \\
\qquad\quad (\mathbf{var}\, \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P; \\
\qquad\quad pc := \mathbf{if}\, b\, \mathbf{then}\, j\, \mathbf{else}\, k) \\
\qquad \ldots \\
\qquad [\!]\, pc = j \longrightarrow B \\
\qquad \ldots \\
\qquad [\!]\, pc = k \longrightarrow C \mathrel{;} pc := i \\
\qquad \ldots \\
\qquad \mathbf{fi} \mathrel{;} \mathit{Poll} \mathrel{;} X \\
\quad \mathbf{fi}
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!]\, \mathit{frameStack} \neq \varnothing \longrightarrow \\
\qquad \mathbf{if} \cdots \\
\qquad [\!]\, pc = i \longrightarrow (\mu Y \bullet A; \\
\qquad\quad (\mathbf{var}\, \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P; \\
\qquad\quad \mathbf{if}\, b \longrightarrow \mathbf{Skip} \\
\qquad\quad [\!]\, \neg\, b \longrightarrow \\
\qquad\qquad pc := k \mathrel{;} \mathit{Poll} \mathrel{;} C; \\
\qquad\qquad pc := i \mathrel{;} \mathit{Poll} \mathrel{;} Y \\
\qquad\quad \mathbf{fi})) \mathrel{;} pc := j \\
\qquad \ldots \\
\qquad [\!]\, pc = j \longrightarrow B \\
\qquad \ldots \\
\qquad [\!]\, pc = k \longrightarrow C \mathrel{;} pc := i \\
\qquad \ldots \\
\qquad \mathbf{fi} \mathrel{;} \mathit{Poll} \mathrel{;} X \\
\quad \mathbf{fi}
\end{array}$$

Figure 5.23: Rule [`while`-loop-intro1]

As a `while` loop may occur with the loop at the end of either conditional branch (since the loop may be created by a `goto` instruction in the Java bytecode), we also provide a similar rule, Rule [`while`-loop-intro2], which introduces the loop in the true branch of the conditional. These two rules are applied on lines 12 and 13 of the algorithm. Rule [`while`-loop-intro2] is presented in Appendix A.

The second type of loop we introduce is the `do-while` loop. A `do-while` loop is similar to a `while` loop, but is distinguished by the fact that the conditional *pc* assignment that causes the loop is at the end of the loop, rather than at the beginning or in the middle. We introduce these loops using Rule [`do-while`-loop-intro], which we omit due to its similarity with Rule [`while`-loop-intro1]; it is also presented in Appendix A. This rule is applied on line 14 of the algorithm. Note that the false branch can never cause the loop in this case, since it will just go to the next instruction. Attempting to redirect it and create the loop with a `goto` instruction would add an instruction within the loop after the conditional, so it would be dealt with as a `while` loop. Therefore, it is not necessary to provide two compilation rules for `do-while` loops.

The final loop structure that we attempt to introduce is that of an infinite loop. An infinite loop may be identified as a block of instructions that ends with a *pc* assignment that causes a jump back to the beginning of the block of instructions. We introduce these loops using Rule [infinite-loop-intro], presented in Appendix A. This rule is applied on line 15 of the algorithm.

After we have attempted to introduce each of the structures for a particular node, we attempt to introduce a sequential composition. This ensures that `if`, `if-else`, `while` and `do-while` structures that occur within conditionals are sequentially composed with the node following them if possible. It also handles cases where sequential compositions occur before loops, preventing them from being introduced in Section 5.3.3 without interfering with the introduction of the loop. Such a case occurs at the $pc = 7$ node in our example.

The requirement for sequential composition to be introduced is the same as in Section 5.3.3: it must be a simple sequential composition from a node with a single outgoing edge to a node with a single incoming edge. Thus we check for a simple sequence on line 16 of Algorithm 4. The sequential composition is then introduced on line 17 if it is a simple sequential composition.

As mentioned earlier, these steps are repeated for each node, working backwards through the control-flow graph of each method. Each of the rules for introducing control flow structures reduces the graph to either a sequential composition graph (Figure 5.1a) or a single node.

Divergent conditionals and infinite loops are the structures whose control-flow graphs are reduced to a single node. The remaining structures are reduced to sequential composition graphs. The reduction of the sequential composition graph depends on which form of node replacement is used to embed the structure in the control-flow graph of the method. There are four cases to consider: root-node replacement (Figure 5.3a), end-node replacement (Figure 5.3b), internal-node replacement (Figure 5.3c) and branch-end replacement (Figure 5.3d).

Replacing the root node of a graph G with a graph H can be viewed as replacing the end node of H with G. Since we are considering the nodes moving backwards through the control flow graph, we always treat this as an end node replacement.

In the cases of end-node replacement and internal-node replacement we can introduce the sequential composition immediately, reducing the graph to a single node.

In the case of branch-end replacement, if some graph H is embedded in a graph G, then reducing H to a sequential composition results in the overall graph having the form of G. This can be

$$\begin{tikzcd}
 & {} \arrow[d] & \\
 & 1 \arrow[dl] \arrow[dr] & \\
a \arrow[dr] & & 3 \arrow[dl] \\
 & 4 &
\end{tikzcd}$$

Figure 5.24: The graph of Figure 5.3d after loop introduction

seen from the example shown in Figure 5.3d, where reducing the `while` loop structure formed by nodes a, b and 4 to a sequential composition yields the graph shown in Figure 5.24. This has the form of a `if-else` conditional (Figure 5.1c). Such structures are introduced on further iterations of the loop over the nodes. Thus, given a structured control-flow graph at the beginning of this stage, the control-flow graph is reduced to a single node, with all the control-flow structures in the method introduced.

In our example, we begin at the $pc = 35$ node, where there are no structures to introduce. The same holds true of the $pc = 28$ and $pc = 32$ nodes (note that the edges coming from them are not simple sequential compositions). An `if-else` conditional is introduced at $pc = 21$, absorbing the $pc = 28$ and $pc = 32$ nodes. The sequential composition from the $pc = 21$ node to the $pc = 35$ node can then be introduced immediately as it is now a simple sequential composition (because it is not at the end of an outer conditional). We then introduce a `while` loop at the $pc = 39$ node (using Rule [`while`-loop-intro2]), and the sequential composition with the $pc = 42$ node is introduced afterwards. Finally, a sequential composition from the $pc = 7$ to the $pc = 39$ node is introduced, collapsing the control flow graph to a single node. The code at $pc = 7$ is then that shown earlier in Figure 5.9.

5.3.5 Resolve Method Calls

When a method is complete, calls to that method can then be resolved. This step begins with the copying of the method into a separate action, so that it can be referenced elsewhere. This is performed as described by Algorithm 5.

Algorithm 5 SeparateCompleteMethods

```
1: $methods \leftarrow \textsc{MethodEntriesAndActionNames}(cs)$
2: for $(methodEntry, methodName) \leftarrow methods$ do
3:     if $\textsc{MethodIsComplete}(methodEntry)$ then
4:         match (if $\cdots$ [] $pc = methodEntry \longrightarrow A \cdots$ fi) in $\textsc{ActionBody}(Running)$ then
5:             apply Law [action-intro]$(methodName,\ A)$
6:         apply Law [copy-rule]$(methodName)$ in reverse
7:     end if
8: end for
```

Algorithm 5 considers each method separately. The method entry point addresses and names that should be used for each method's action are extracted from the class information cs by the function METHODENTRIESANDACTIONNAMES(*cs*), on line 1. The name used for the action does not have an effect upon the correctness of the strategy, provided it is unique. We adopt the icecap convention and form the name of this action from the name of the class to which the method belongs and the name of the method, concatenated together with an underscore. We then iterate over each method entry point, *methodEntry*, and method action name, *methodName*, as specified by the loop on line 2.

For each method, we determine if the sequence of actions beginning at *methodEntry* is complete, on line 3. This involves a simple syntactic check that each conditional branch ends in a return instruction or a recursion. For methods that are complete, the sequence of actions for the method are placed in a separate action, which is introduced using Law [action-intro] on line 5. The body of the action to be introduced is obtained from the sequence of instructions at the $pc = methodEntry$ branch of the choice in *Running*, as specified by the pattern match on line 4. Once the method action has been introduced, the sequence of actions at the method's entry point in the *Running* action is replaced with a reference to the newly introduced action by applying Law [copy-rule] on line 6.

In our example, the method `f()` of the `TPK` class, which starts at $pc = 43$, is complete on the first iteration of the loop on line 3 of Algorithm 1. This can be seen in Figure 5.8. This method is complete because it consists of a straight sequence of instructions ending with *HandleAreturnEPC*, which represents the `areturn` instruction. The sequence of instructions at $pc = 43$ is copied into the action *TPK_f*, which can be seen in Figure 5.10. The $pc = 43$ branch is replaced with a call to *TPK_f*.

After all the complete methods have been copied into separate actions, calls to those methods are resolved. This is performed as described by Algorithm 6. In this algorithm, while we indicate the parameters supplied to a rule in brackets after the rule name, as in previous algorithms, we use the word **to** to indicate which part of the *Running* action the rule is applied to. In all previous algorithms, the laws are applied to the whole *Running* action, and so we omit the **to** clause.

The algorithm operates on each unresolved method call present in the choice in *Running*, constructing a list of them on line 1. This list is obtained using the UNRESOLVEDMETHODCALLS function, which finds all the branches in the choice in *Running* ending with a method invocation instruction (*HandleInvokespecialEPC*, *HandleInvokestaticEPC* or *HandleInvokevirtualEPC*). The presence of such a method call at the end of a sequence of actions indicates that method call has not yet been resolved, since it would be followed by a *pc* assignment and possibly other actions if it had been resolved. An example of an unresolved method call can be seen in the $pc = 14$ branch of Figure 5.8, reproduced below.

$$\begin{array}{l} \ldots \\ \square\ pc = 14 \longrightarrow HandleAstoreEPC(2)\ ;\ pc := 15\ ;\ Poll\ ;\ HandleAloadEPC(1); \\ \quad pc := 16\ ;\ Poll\ ;\ \{pc = 16\}\ ;\ HandleInvokevirtualEPC(36) \\ \ldots \end{array}$$

This ends with the action *HandleInvokevirtualEPC*(36), which handles the `invokevirtual` instruction for the constant pool index 36. It is unresolved because it does not have any *pc* assignment or other actions following it.

We iterate over the set of method calls in the for loop beginning on line 2. The program address for the sequence of instructions ending in the unresolved method call is referred to as *methodCallAddress*, and the method call action at the end of the sequence is referred to as *methodCall*.

Algorithm 6 ResolveMethodCalls

```
 1: methodCalls ← UnresolvedMethodCalls
 2: for methodCallAddress ← methodCalls do
 3:     methodCall ← MethodCallAction(methodCallAddress)
 4:     if IsResolvable(methodCall) then
 5:         try
 6:             apply Rule [refine-invokespecial] to methodCall
 7:             apply Rule [refine-invokestatic] to methodCall
 8:             apply Rule [refine-invokevirtual] to methodCall
 9:         end try
10:         if HasStaticDispatch(methodCall) then
11:             try
12:                 apply Rule [resolve-special-method] to methodCall
13:                 apply Rule [resolve-normal-method](methodCallAddress)
14:             end try
15:             try
16:                 apply Rule [CheckSynchronizedInvoke-sync-refinement] to target
17:                 apply Rule [CheckSynchronizedInvoke-nonsync-refinement] to target
18:             end try
19:         else
20:             for target ←Targets(methodCall) do
21:                 apply Rule [resolve-normal-method-branch](methodCall, target)
22:                 try
23:                     apply Rule [CheckSynchronizedInvoke-sync-refinement] to target
24:                     apply Rule [CheckSynchronizedInvoke-nonsync-refinement] to target
25:                 end try
26:             end for
27:             apply Rule [virtual-method-call-dist] to methodCall
28:         end if
29:         if HasSimpleSequence(methodCallAddress) then
30:             apply Rule [sequence-intro](methodCallAddress)
31:         end if
32:     end if
33: end for
```

For each method call that needs resolving, we check if it can be resolved at this point in the compilation strategy. This is performed on line 4, where the boolean IsResolvable(*methodCall*) is checked. IsResolvable(*methodCall*) is true if all the targets of the method call *methodCall* are either special methods or non-special methods that are already complete and have been separated into their own actions (as described in Algorithm 5).

If the method call is resolvable, we replace the action that handles the method invocation instruction with an action that pops the arguments for the method from the stack and handles invocation of the specific method referenced by the instruction. This is handled slightly differently for each of the method invocation instructions in our bytecode subset, so we have three rules for performing this transformation, one for each instruction: Rule [refine-invokestatic], Rule [refine-invokespecial] and Rule [refine-invokevirtual]. They produce slightly different sequences of actions due to the differences in the semantics of the method invocation instructions, described in Section 4.3.4. They are applied in the try block beginning on line 5 of Algorithm 6.

In Figure 5.25 we show Rule [refine-invokestatic], which handles `invokestatic` instructions. This rule, as with other rules in this section, is applied to an action beginning with an assumption on the value of *pc*. This assumption is present before all actions that handle method invocation instructions since it is introduced during bytecode expansion (Section 5.3.2). Rule [refine-

Rule [refine-invokestatic].

$$\begin{array}{l} \{pc = i\}; \\ HandleInvokestaticEPC(cpi) \end{array} \sqsubseteq_A \begin{array}{l} \{pc = i\}\ ;\ \mathbf{var}\, poppedArgs : \mathrm{seq}\, Word \bullet \\ \quad \big(\exists\, argsToPop? == methodArguments\ m \bullet \\ \qquad InterpreterStackFrameInvoke\big); \\ \quad Invoke(c, m, poppedArgs) \end{array}$$

where $m : MethodID$ and $c : ClassID$ are such that

$$\begin{array}{l} \exists\, c_0 : Class \mid c_0 \in \mathrm{ran}\, cs \bullet \\ \quad (\exists\, m_0 : MethodID \mid m_0 \in \mathrm{dom}\, c_0.methodEntry \bullet \\ \qquad i \in c_0.methodEntry\ m_0 \mathrel{..} c_0.methodEnd\ m_0) \\ \quad cpi \in methodRefIndices\ c_0 \land c_0.constantPool\ cpi = MethodRef\ (c, m). \end{array}$$

Figure 5.25: Rule [refine-invokestatic]

invokestatic] refines *HandleInvokestaticEPC*(*cpi*) to an action that pops the method's arguments from the stack using the *InterpreterStackFrameInvoke* operation and then behaves as the *Invoke* action described in Section 4.3.

The method handled by *Invoke* is identified by a class identifier, c, and a method identifier, m. These identifiers are determined from the *cpi* parameter passed to *HandleInvokestaticEPC*, which is an index into the *constantPool* of the current class information. To determine the identifiers, we first determine the current class information, c_0, which is the class in *cs* that contains a method, m_0, whose bytecode spans over the current *pc* value, i. Within c_0, the *constantPool* entry at *cpi* must be a *MethodRef*. The c and m values of the method to be invoked are those contained in the *MethodRef*. These are passed to *Invoke*, along with the arguments popped from the stack, *poppedArgs*.

Rule [refine-invokespecial] is similar to Rule [refine-invokestatic]. It provides for popping an additional `this` argument from the stack, and enforces the rules given in the *CheckSuperclass* schema for selecting the class identifier.

Rule [refine-invokevirtual] (Figure 5.26) refines an `invokevirtual` method call, introducing a choice over all the possible targets of the call. It replaces the action *HandleInvokevirtualEPC* with an action that pops the arguments of the function from the stack using the data operation *InterpreterStackFrameInvoke*, and then makes a choice of which method to invoke using the class of the `this` argument for the method. The set of possible targets classes for the choice are those that are both subclasses of the class referenced by the `invokevirtual` instruction and in the *instCS* set of instantiated classes. The method invocations are left as references to the *Invoke* actions, to be resolved later in Algorithm 6. The assumption on the value of *pc* is converted to an assumption on the *storedPC* value of the current stack frame, and distributed into each of the branches of the choice, so it can be handled separately for each branch.

The class identifiers used in the choice are determined by looking up the constant pool index, *cpi*, as for Rule [refine-invokestatic], to obtain a class identifier, c, and method identifier, m.

Rule [refine-invokevirtual]**.** Given $i : \mathit{ProgramAddress}$,

$$
\begin{array}{l}
\{pc = j\}; \\
\mathit{HandleInvokevirtualEPC}(cpi)
\end{array}
\sqsubseteq_A
\begin{array}{l}
\mathbf{var}\ \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet \\
\big(\exists\ \mathit{argsToPop?} == \mathit{methodArguments}\ m \bullet \\
\quad \mathit{InterpreterStackFrameInvoke}\big); \\
\mathit{getClassIDOf}!(\mathit{head}\ \mathit{poppedArgs})?cid \longrightarrow \\
\mathbf{if}\ cid = c_1 \longrightarrow \\
\quad \{(\mathit{last}\ \mathit{frameStack}).\mathit{storedPC} = j + 1\}; \\
\quad \mathit{Invoke}(c_1, m, \mathit{poppedArgs}) \\
\ldots \\
\ [\!]\ cid = c_n \longrightarrow \\
\quad \{(\mathit{last}\ \mathit{frameStack}).\mathit{storedPC} = j + 1\}; \\
\quad \mathit{Invoke}(c_n, m, \mathit{poppedArgs}) \\
\mathbf{fi}
\end{array}
$$

where $m : \mathit{MethodID}$ and $c_1, \ldots, c_n : \mathit{ClassID}$ are such that

$$
\begin{array}{l}
\exists\ c_0 : \mathit{Class};\ m_0 : \mathit{MethodID} \mid c_0 \in \mathrm{ran}\ cs \land m_0 \in \mathrm{dom}\ c_0.\mathit{methodEntry} \bullet \\
\quad cpi \in \mathit{methodRefIndices}\ c_0 \land \\
\quad j \in c_0.\mathit{methodEntry}\ m_0 \mathrel{..} c_0.\mathit{methodEnd}\ m_0 \land \\
\quad \exists\ c : \mathit{ClassID} \bullet c_0.\mathit{constantPool}\ cpi = \mathit{MethodRef}\ (c, m) \land \\
\quad \{x : \mathit{ClassID} \mid (x, c) \in \mathit{subclassRel}\ cs \land x \in \mathit{instCS}\} = \{c_1, \ldots, c_n\}
\end{array}
$$

Figure 5.26: Rule [refine-invokevirtual]

The identifier m determines which method should be invoked, but the class of the method to be invoked is determined from the class of the `this` object popped from the stack. Since Java bytecode verification ensures that the class is assignable to c, we need only consider the identifiers of subclasses of c, and these are further constrained by instCS, the classes that are actually instantiated in the program, to obtain the list of targets: $c_1, \ldots, c_n$.

After one of the above rules is applied, the method invocation is resolved by transforming the *Invoke* action to the behaviour of the method being invoked. A *pc* assignment is also introduced after the method's behaviour so that it can be sequentially composed with the instructions after the method call. This is performed separately depending on whether the method call is performed with static dispatch (`invokespecial` and `invokestatic`) or dynamic dispatch (`invokevirtual`), since each of the possible targets must be considered in the dynamic dispatch case. The type of dispatch is thus checked in the if statement on line 10 of Algorithm 6. This is a simple syntactic check as to whether the method call has a *getClassIDOf* communication and a choice over the possible targets. In the static dispatch case, there are two rules, applied in another try block on line 11: Rule [resolve-special-method], which handles resolution of special methods, and Rule [resolve-normal-method], which handles resolution of non-special methods.

Rule [resolve-special-method], shown in Figure 5.27, operates by simply replacing the call to the *Invoke* action with actions that specify the behaviour for the special method, and introducing a *pc* assignment after those actions. This collapses the choice in the definition of the *Invoke* action. The assumption on the value of *pc* is also eliminated, since it is no longer needed. This is also the case for the other method-resolution rules applied in the try block on line 11. The actions that define the behaviour of the special method are identified by the syntactic function *specialMethodAction*, which is defined by Table 5.2. It determines which behaviour should be used based on the class and method identifiers passed to *Invoke*.

Rule [resolve-special-method]**.** If c, m match one of the rows of Table 5.2, then

$$\begin{array}{l} \{pc = i\} \mathbin{;} (\mathbf{var}\, poppedArgs : \mathrm{seq}\, Word \bullet \\ \big(\exists\, argsToPop? == e \bullet \\ \quad InterpreterStackFrameInvoke\big); \\ Invoke(c, m, poppedArgs)) \end{array} \sqsubseteq_A \begin{array}{l} (\mathbf{var}\, poppedArgs : \mathrm{seq}\, Word \bullet \\ \big(\exists\, argsToPop? == e \bullet \\ \quad InterpreterStackFrameInvoke\big); \\ specialMethodAction(c, m)); \\ pc := i + 1 \end{array}$$

where *specialMethodAction* is the syntactic function defined by Table 5.2.

Figure 5.27: Rule [resolve-special-method]

Rule [resolve-normal-method], shown in Figure 5.28, resolves non-special methods by unrolling the loop in *Running* to sequence the method call with the action defining the method's behaviour. The entry point of the method is obtained from the class information for the method, which is determined as described by the data operation *ResolveMethod*. The first proviso of the rule requires that the nonemptiness of the *frameStack* is not affected by the instructions before the method invocation, as with previous compilation rules in this stage. The second proviso of the rule ensures that the entry point, k, is that given in the class information provided by *ResolveMethod* for the class identifier c and method identifier m.

The action containing the behaviour of the method is M, at the $pc = k$ branch of the choice in *Running*. The third proviso requires that the execution of M must result in the top stack frame being popped and the pc being set to the value stored in the next stack frame. This is needed to ensure that the method can be sequenced with the behaviour after it. It is true for all complete methods, since the return instructions establish the required property and any property may be assumed to hold after an infinite loop.

Rule [resolve-normal-method] applies only to those class and method identifiers that are not handled by Rule [resolve-special-method]. Because of this, Rule [resolve-normal-method] collapses the choice in the *Invoke* action, replacing it with *CheckSynchronizedInvoke* and the data operation *InterpreterNewStackFrame*, sequenced with the action *Poll* and the method action, M, defining the method's behaviour. An assignment is placed after M to set pc to the address of the next instruction.

After the resolution of a non-special method, a *CheckSynchronizedInvoke* action is left before the method call. We eliminate this action by the application of Rule [*CheckSynchronizedInvoke*-sync-refinement] and Rule [*CheckSynchronizedInvoke*-nonsync-refinement] in the try block beginning on line 15. These refine *CheckSynchronizedInvoke*, collapsing the choice in that action using the arguments passed to it. Rule [*CheckSynchronizedInvoke*-sync-refinement] results in a communication on the *takeLock* channel if the resolved method is synchronized and not static. For methods that are static or not synchronized, Rule [*CheckSynchronizedInvoke*-nonsync-refinement] refines *CheckSynchronizedInvoke* to **Skip**.

In the case of dynamic dispatch, we iterate over each branch of the choice over the method's targets, in the loop beginning on line 20, using the function TARGETS(*methodCall*) to obtain a list of the possible targets of the method call *methodCall*. For each target, we apply Rule [resolve-normal-method-branch]. This is similar to Rule [resolve-normal-method], but operates over only a single branch of the choice of targets. We omit this rule due to its similarity with the rule

Conditions on c and m	$\mathit{specialMethodAction}(c, m)$
$(c, \mathit{setPriorityCeilingClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{setPriorityCeilingID}$	$\mathit{setPriorityCeiling}!(\mathit{methodArgs}\ 1)!(\mathit{methodArgs}\ 2) \longrightarrow \mathit{setPriorityCeilingRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{aperiodicEventHandlerClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{releaseAperiodicID}$	$\mathit{releaseAperiodic}!(\mathit{methodArgs}\ 1) \longrightarrow \mathit{releaseAperiodicRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{writeClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{writeID}$	$\mathit{output}!(\mathit{methodArgs}\ 1) \longrightarrow \mathbf{Skip}$
$(c, \mathit{readClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{readID}$	$\mathit{input}?\mathit{value} \longrightarrow \left(\mathit{InterpreterPush} \setminus (pc, pc')\right)$
$(c, \mathit{managedSchedulableClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{registerID}$	$\mathit{register}!\mathit{thread}!(\mathit{head}\ \mathit{methodArgs}) \longrightarrow \mathit{registerRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{managedMemoryClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{enterPrivateMemoryHelperID}$	$\mathit{enterPrivateMemory}!\mathit{thread}!(\mathit{methodArgs}\ 1) \longrightarrow \mathit{enterPrivateMemoryRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{managedMemoryClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{executeInAreaOfHelperID}$	$\mathit{executeInAreaOf}!\mathit{thread}!(\mathit{methodArgs}\ 1) \longrightarrow \mathit{executeInAreaOfRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{managedMemoryClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{executeInOuterAreaHelperID}$	$\mathit{executeInOuterArea}!\mathit{thread} \longrightarrow \mathit{executeInOuterAreaRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{managedMemoryClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{exitMemoryID}$	$\mathit{exitMemory}!\mathit{thread} \longrightarrow \mathit{exitMemoryRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{aperiodicEventHandlerClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{initAPEHID}$	$\mathit{initAPEH}!\mathit{thread}!(\mathit{seqTo5Tuple}\ \mathit{methodArgs}) \longrightarrow \mathit{initAPEHRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{periodicEventHandlerClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{initPEHID}$	$\mathit{initPEH}!\mathit{thread}!(\mathit{seqTo7Tuple}\ \mathit{methodArgs}) \longrightarrow \mathit{initPEHRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{oneShotEventHandlerClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{initOSEHAbsID}$	$\mathit{initOSEHAbs}!\mathit{thread}!(\mathit{seqTo6Tuple}\ \mathit{methodArgs}) \longrightarrow \mathit{initOSEHAbsRet} \longrightarrow \mathbf{Skip}$
$(c, \mathit{oneShotEventHandlerClass}) \in \mathit{subclassRel}\ \mathit{cs}$ $\land\ m = \mathit{initOSEHRelID}$	$\mathit{initOSEHRel}!\mathit{thread}!(\mathit{seqTo6Tuple}\ \mathit{methodArgs}) \longrightarrow \mathit{initOSEHRelRet} \longrightarrow \mathbf{Skip}$

Table 5.2: The syntactic function $\mathit{specialMethodAction}(c, m)$

Rule [resolve-normal-method]**.** Given $i : ProgramAddress$, if

- $$\{frameStack \neq \varnothing\} \;;\; A \\ = \\ \{frameStack \neq \varnothing\} \;;\; A \;;\; \{frameStack \neq \varnothing\},$$

- $methodID = m \wedge classID = c \Rightarrow \mathbf{pre}\, ResolveMethod$ and there is $classInfo : Class$ such that

  $$\{methodID = m \wedge classID = c\} \;;\; \big(ResolveMethod\big) \\ = \\ \{methodID = m \wedge classID = c\} \;;\; \big(ResolveMethod\big); \\ \{class = classInfo \wedge class.methodEntry\, m = k\},$$

- for any $x : ProgramAddress$,

  $$\{(last\,(front\,frameStack)).storedPC = x\} \;;\; M \\ = \\ \{(last\,(front\,frameStack)).storedPC = x\} \;;\; M \;;\; \{pc = x\},$$

- m and c do not match any of the conditions in Table 5.2,

then,

```
$\mu X \bullet$
  $\mathbf{if}\, frameStack = \varnothing \longrightarrow \mathbf{Skip}$
  $[\!]\, frameStack \neq \varnothing \longrightarrow$
    $\mathbf{if} \cdots$
    $[\!]\, pc = i \longrightarrow A \;;\; \{pc = j\};$
      $\mathbf{var}\, poppedArgs : \mathrm{seq}\, Word \bullet$
      $\big(\exists\, argsToPop? == e \bullet$
        $InterpreterStackFrameInvoke\big);$
      $Invoke(c, m, poppedArgs)$
    $[\!]\, pc = k \longrightarrow M$
    $\cdots$
    $\mathbf{fi} \;;\; Poll \;;\; X$
  $\mathbf{fi}$
```

$\sqsubseteq_A$

```
$\mu X \bullet$
  $\mathbf{if}\, frameStack = \varnothing \longrightarrow \mathbf{Skip}$
  $[\!]\, frameStack \neq \varnothing \longrightarrow$
    $\mathbf{if} \cdots$
    $[\!]\, pc = i \longrightarrow A;$
      $(\mathbf{var}\, poppedArgs : \mathrm{seq}\, Word \bullet$
      $\big(\exists\, argsToPop? == e \bullet$
        $InterpreterStackFrameInvoke\big);$
      $CheckSynchronizedInvoke($
        $classInfo, m, poppedArgs);$
      $\big(InterpreterNewStackFrame[$
        $classInfo / class?,$
        $m / methodID?,$
        $poppedArgs / methodArgs?]\big);$
      $Poll \;;\; M) \;;\; pc := j + 1$
    $[\!]\, pc = k \longrightarrow M$
    $\cdots$
    $\mathbf{fi} \;;\; Poll \;;\; X$
  $\mathbf{fi}$
```

Figure 5.28: Rule [resolve-normal-method]

previously presented. It can be found in Appendix A. Note that, since all our special methods are static, none of them can occur as the target of a `invokevirtual` instruction, so we do not need to handle special methods in the dynamic dispatch case. When this has been applied, we apply Rule [*CheckSynchronizedInvoke*-sync-refinement] and Rule [*CheckSynchronizedInvoke*-nonsync-refinement to the target, as in the case of static dispatch. After each of the targets has been resolved, the *pc* assignment is moved outside the choice by an application of Rule [virtual-method-call-dist] on line 27, which distributes the action out of the conditional and moves it outside the variable block surrounding the conditional.

In both the case of a single target and the case of multiple targets, we attempt to introduce a sequential composition with the instructions after the method call. This is done on lines 29 to 31 of Algorithm 6 in the same way as in Algorithm 4. It may not be possible to introduce the sequential composition at this point if, for example, a method call occurs at the end of a conditional branch, since we must wait until the conditional has been introduced before the sequential composition can be introduced.

As an example of method call resolution, we consider the `invokestatic` instruction at $pc = 23$. Before method call resolution this appears in the choice in *Running* as shown below.

$$\begin{array}{l}\dots\\ \mathrel{[\!]}\ pc = 23 \longrightarrow \{pc = 23\}\ ;\ \ \mathit{HandleInvokestatic}(46)\\ \dots\end{array}$$

The *pc* value 23 is between the *methodEntry* and *methodEnd* values for *handleAsyncEvent* in the *Class* information *TPK*, shown in Figure 5.5. The constant pool index 46 is thus looked up in *TPK*'s *constantPool*, yielding a *MethodRef* containing the class identifier *TPKClassID* and method identifier f. Since the instruction being handled is an `invokestatic` instruction, there is only a single target, which is the method referenced by these identifiers. That method is the `f()` method of `TPK`, whose entry point is at $pc = 43$. There is a straight sequence of instructions at this entry point, ending with an `areturn` instruction. Thus, it has already been sequenced together when method resolution occurs for the first time, and separated into a method action *TPK_f*, which can be seen in Figure 5.10. This method call can thus be resolved.

The *HandleInvokestatic*(46) action is refined using Rule [refine-invokestatic]. After applying this rule, the sequence of actions starting at $pc = 23$ has the following form.

$$\begin{array}{l}\dots\\ \mathrel{[\!]}\ pc = 23 \longrightarrow \{pc = 23\}\ ;\ \ \mathbf{var}\, \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet\\ \quad\big(\exists\, \mathit{argsToPop}? == \mathit{methodArguments}\, f \bullet\\ \qquad\quad \mathit{InterpreterStackFrameInvoke}\big);\\ \quad \mathit{Invoke}(\mathit{TPKClassID}, f, \mathit{poppedArgs})\\ \dots\end{array}$$

After refining the action with Rule [refine-invokestatic], we resolve the method call using Rule [resolve-normal-method], since `f()` is not a special method. The second proviso of this rule ensures that it is applied with $k = 43$, since *TPK* matches the class identifier *TPKClassID* and contains information for the method identifier f. The third proviso is met, since *TPK_f* ends with *HandleAreturnEPC*, which pops the last frame from the *frameStack* and sets *pc* to the stored value. After the application of Rule [resolve-normal-method], the sequence of actions has the form below, with the method invocation sequenced with the *TPK_f* action and an assignment $pc := 24$.

$$\begin{array}{l}
\ldots \\
[\!]\ pc = 23 \longrightarrow (\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \\
\quad \big(\exists\ argsToPop? == methodArguments\ m \bullet InterpreterStackFrameInvoke\big); \\
\quad \big(InterpreterNewStackFrame[\\
\qquad TPK/class?, f/methodID?, poppedArgs/methodArgs?]\big); \\
\quad Poll\ ;\ \ TPK\_f)\ ;\ \ pc := 24 \\
\ldots
\end{array}$$

A sequential composition can then be introduced with the instructions at $pc = 24$, to yield the code in Figure 5.11.

As mentioned previously, the resolution of methods calls and introduction of loops and conditionals is performed in a loop until all the methods have been separated into their own actions. After that, the remaining uses of the program counter in the main actions of *Thr* are eliminated as described in the next section.

5.3.6 Refine Main Actions

After the control flow of each method has been introduced and each method has been separated into its own method action, the only remaining uses of *pc* are to select a method action when a method is executed in response to a request from the *Launcher*. This occurs in the *MainThread* and *Started* actions, where a call to *Running* follows a call to *StartInterpreter*. To remove these final uses of *pc*, we replace *Running* with a call to a new action, *ExecuteMethod*, which chooses a method action based on a class and method identifier. This performed as specified in Algorithm 7.

Algorithm 7 REFINEMAINACTIONS

```
1: apply Law [copy-rule]($Running$) to ACTIONBODY($MainThread$)
2: apply Law [copy-rule]($Running$) to ACTIONBODY($Started$)
3: apply Rule [$StartInterpreter$-$Running$-refinement] to ACTIONBODY($MainThread$)
4: apply Rule [$StartInterpreter$-$Running$-refinement] to ACTIONBODY($Started$)
5: match  $executeMethod?t : (t = thread)?c?m?a$  in ACTIONBODY($Started$) then
                $\longrightarrow (A)(c, m, a)$
6:     apply Law [action-intro]($ExecuteMethod$, $A$)
7: apply Law [copy-rule]($ExecuteMethod$) to ACTIONBODY($MainThread$)
8: apply Law [copy-rule]($ExecuteMethod$) to ACTIONBODY($Started$)
```

Algorithm 7 differs from previous algorithms in that it does not operate purely upon the *Running* action. We instead refine the composition of *StartInterpreter* and *Running* in *MainThread* and *Started*. First, Law [copy-rule] is applied on lines 1 and 2 to replace the call to *Running* with its body in *MainThread* and *Started*. Then, we apply Rule [*StartInterpreter-Running*-refinement], shown in Figure 5.29, on lines 3 and 4. This refines the composition of *Started* with the body of *Running* in *MainThread* and *Started*.

When we apply Rule [*StartInterpreter-Running*-refinement], we first introduce an assumption stating that the *frameStack* is nonempty before execution of *StartInterpreter*. This is true in both the places that the rule is applied, since the *frameStack* is initially empty and no stack

Rule [*StartInterpreter*-*Running*-refinement]**.** If $(c_1, m_1), \ldots, (c_n, m_n)$ are the only $ClassID \times MethodID$ values such that $classID = c_i \land methodID = m_i \Rightarrow \mathbf{pre}\, ResolveMethod$, and for each $i \in \{1 \mathrel{..} n\}$, there exists $classInfo_i : Class$ and $entry_i : ProgramAddress$ such that,

$$\begin{array}{l}
\{classID = c_i \land methodID = m_i\} \;;\; ResolveMethod \\
= \\
\{classID = c_i \land methodID = m_i\} \;;\; ResolveMethod; \\
\qquad \{class = classInfo_i \land classInfo_i.methodEntry\ m_i = entry_i\},
\end{array}$$

and, for each $i \in \{1 \mathrel{..} n\}$,

$$\begin{array}{l}
\{\#\, frameStack = 1\} \;;\; M_i \\
= \\
\{\#\, frameStack = 1\} \;;\; M_i \;;\; \{framestack = \varnothing\},
\end{array}$$

then,

$$\begin{array}{lcl}
\begin{array}{l}
\{frameStack = \varnothing\}; \\
StartInterpreter \;;\; Poll \;;\; \mu X \bullet \\
\quad \mathbf{if}\, frameStack = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!]\; framestack \neq \varnothing \longrightarrow \\
\qquad \mathbf{if}\, pc = entry_1 \longrightarrow M_1 \\
\qquad \ldots \\
\qquad [\!]\; pc = entry_n \longrightarrow M_n \\
\qquad \mathbf{fi} \;;\; Poll \;;\; X \\
\quad \mathbf{fi}
\end{array}
& \sqsubseteq_A &
\begin{array}{l}
executeMethod?t : (t = thread)?c?m?a \longrightarrow \\
(\mathbf{val}\, classID : ClassID; \\
\mathbf{val}\, methodID : MethodID; \\
\mathbf{val}\, methodArgs : \operatorname{seq} Word \bullet \\
\mathbf{if}\, (classID, methodID) = (c_1, m_1) \longrightarrow \\
\quad InterpreterNewStackFrame[\\
\qquad classInfo_1 / class?, \\
\qquad m_1 / methodID?] \;;\; Poll \;;\; M_1 \\
\ldots \\
[\!]\; (classID, methodID) = (c_n, m_n) \longrightarrow \\
\quad InterpreterNewStackFrame[\\
\qquad classInfo_n / class?, \\
\qquad m_n / methodID?] \;;\; Poll \;;\; M_n \\
\mathbf{fi})(c, m, a) \;;\; Poll
\end{array}
\end{array}$$

Figure 5.29: Rule [*StartInterpreter*-*Running*-refinement]

frames have been created at the point when *MainThread* and *Started* occur. The *Running* action causes the *stackFrame* to be empty after its execution, so the *stackFrame* is also empty when the *MainThread* and *Started* actions loop to allow *StartInterpreter* to be executed again.

Rule [*StartInterpreter-Running*-refinement] collects all the class and method identifiers that are resolved by the data operation *ResolveMethod*. It expands the definition of *StartInterpreter*, and introduces a choice over these class and method identifiers, comparing them to the identifiers communicated on the *executeMethod* channel.

Within each branch of the choice for a class identifier c_i and method identifier m_i, a new stack frame is first created by the *InterpreterNewStackFrame* operation, using the class information, $classInfo_i$, provided by *ResolveMethod* for c_i and m_i. The branch then behaves as the method action in *Running* that corresponds to the method entry point, $entry_i$, associated with m_i in $classInfo_i$. Note that the mapping from class and method identifiers to class information and method entries is not necessarily injective, since inherited methods share the same class information and bytecode instructions. The choice over class and method identifiers is wrapped in a value parameter block, since it forms the body of the *ExecuteMethod* action.

After Rule [*StartInterpreter-Running*-refinement] has been applied, the *ExecuteMethod* action is introduced in the *Thr* process using Law [action-intro], on line 6 of Algorithm 7. The body of *ExecuteMethod*, introduced in *MainThread* and *NotStarted* by Rule [*StartInterpreter-Running*-refinement], is then replaced with a call to *ExecuteMethod* by application of Law [copy-rule], on lines 7 and 8. This results in *MainThread* and *Started* having the form shown previously in Figure 5.13.

5.3.7 Remove *pc* From State

After *MainThread* and *Started* have been refined, *pc* is no longer used by *Thr*, and so we can remove it from the state of *Thr*, as specified in Algorithm 8. This algorithm operates over the *Thr* process as a whole, since *pc* must be removed from every action in *Thr* simultaneously.

Algorithm 8 REMOVEPCFROMSTATE

```
1: apply Law [forwards-data-refinement]($InterpreterStateEPC$, $CI$)
2: exhaustively apply Law [seq-unitl]
3: apply Law [process-param-elim]($bc$)
4: apply Law [process-param-elim]($instCS$)
```

Algorithm 8 begins with the application of Law [forwards-data-refinement] at line 1. This law describes a standard *Circus* data refinement between processes, in which a coupling invariant is defined to describe the relationship between the old state of the process and the new state of the process. We characterise the refinement by providing the new process state and the coupling invariant. In this case, the relation defined by the coupling invariant is a function, so the actions of the new process can be calculated from the actions of the old process. Thus, the new state and coupling invariant are sufficient to uniquely characterise the data refinement.

For our refinement, the new state is *InterpreterStateEPC*, shown below. It is similar to *InterpreterState*, but the *pc* component is removed. The *frameStack* also has a different type, being a sequence of *StackFrameEPC* structures, which are similar to *StackFrame*, but without the *storedPC* component, since that is only used for storing a value from *pc*. The invariant of *InterpreterStateEPC* is the same as for *InterpreterState*, but without the requirement that the *currentClass* and stack frame *frameClass* values be consistent with the *pc* value.

$$\begin{array}{|l}
\textit{InterpreterStateEPC} \\
\hline
\textit{frameStack} : \operatorname{seq} \textit{StackFrameEPC} \\
\textit{currentClass} : \textit{Class} \\
\hline
\textit{frameStack} \neq \varnothing \Rightarrow \textit{currentClass} = (\textit{last}\,\textit{frameStack}).\textit{frameClass} \\
\hline
\end{array}$$

The coupling invariant, *CI*, for the refinement from *InterpreterState* to *InterpreterStateEPC* is shown below, with *InterpreterStateEPC* decorated with $_1$ to distinguish its components. *CI* equates the *currentClass* components of the two schemas, since they are unaffected. The *frameStack* components are declared to have the same domain, and each *StackFrame* in the *frameStack* is mapped onto a *StackFrameEPC* with the same *localVariables*, *operandStack*, *frameClass*, and *stackSize* values. The *pc* and *storedPC* values of *InterpreterState* are discarded, since they are not present in *InterpreterStateEPC*.

$$\begin{array}{|l}
\textit{CI} \\
\hline
\textit{InterpreterState} \\
\textit{InterpreterStateEPC}_1 \\
\hline
\textit{currentClass} = \textit{currentClass}_1 \\
\operatorname{dom} \textit{frameStack} = \operatorname{dom} \textit{frameStack}_1 \\
\forall\, i : \operatorname{dom} \textit{frameStack} \bullet \\
\qquad (\textit{frameStack}\ i).\textit{localVariables} = (\textit{frameStack}_1\ i).\textit{localVariables} \land \\
\qquad (\textit{frameStack}\ i).\textit{operandStack} = (\textit{frameStack}_1\ i).\textit{operandStack} \land \\
\qquad (\textit{frameStack}\ i).\textit{frameClass} = (\textit{frameStack}_1\ i).\textit{frameClass} \land \\
\qquad (\textit{frameStack}\ i).\textit{stackSize} = (\textit{frameStack}_1\ i).\textit{stackSize} \\
\hline
\end{array}$$

This data refinement has the effect of removing *pc* from each of the data operations in *Thr*. This effect is minimal for most data operations, since their *pc* updates have already been extracted. However, *InterpreterStackFrameInvoke* no longer stores the current *pc* value in the *storedPC* component of the topmost stack frame, and *InterpreterNewStackFrame* does not set the *pc* value. Additionally, the method return operations *InterpreterAreturn* and *IntepreterReturn* do not set the value of *pc* using the *storedPC* value of the previous stack frame.

The *pc* assignments introduced between bytecode instructions during the strategy are also affected by the data refinement. The data refinement removes *pc* from the assignments, leaving only their effect on the other components of the state. Since the assignments leave all other state components unchanged, the data-refined *pc* assignments have no effect, making them equivalent to **Skip**. These **Skip** actions are then eliminated by applying Law [seq-unitl] wherever possible, on line 2 of Algorithm 8.

Finally, we eliminate the *bc* and *instCS* parameters to the process, since they are also no longer needed, using application of Law [process-param-elim], on lines 3 and 4. This completes the refinement of $\mathit{Thr}(bc, cs, instCS, t)$ into $\mathit{ThrCF}_{bc,cs}(cs, t)$, referenced in Theorem 5.3.1, which has its control flow introduced and does not include *pc* in its state. The next stage of the strategy operates on $\mathit{ThrCF}_{bc,cs}(cs, t)$ to eliminate the *frameStack*.

Algorithm 9 Elimination of Frame Stack

```
1: RemoveLauncherReturns
2: LocaliseStackFrames
3: IntroduceVariables
4: RemoveFrameStackFromState
```

5.4 Elimination of Frame Stack

The second stage of the compilation strategy eliminates the *frameStack* from the state of each thread's process, $ThrCF_{bc,cs}(cs, t)$. The information stored in the stack frames on *frameStack* is transferred into variables representing the local variables and operand stack slots for each method. The operations representing the bytecode instructions are refined to operations over these variables. This refines $ThrCF_{bc,cs}(cs, t)$ to the $CThr_{bc,cs}(t)$ process described in Section 4.4.1, so this stage may be summarised by the following theorem.

Theorem 5.4.1 (Elimination of Frame Stack)**.**

$$ThrCF_{bc,cs}(cs, t) \sqsubseteq CThr_{bc,cs}(t)$$

In this stage, we operate mainly on the method actions introduced in the previous stage. Algorithm 9 describes the strategy for transforming those actions to introduce variables and eliminate the *frameStack*. It begins on line 1 by refining the return instructions that occur at the end of each method action to remove the *CheckLauncherReturn* actions that occur in those instructions, resolving the check of whether *frameStack* is empty. This removes the only remaining use of *frameStack* as a whole, enabling us to consider the stack frames for each method individually. We introduce a variable in each method that contains its stack frame, on line 2 of the algorithm, and convert the operations of the method to operate over the new variable rather than the global *frameStack*. Afterwards, on line 3, we perform local data refinements to convert the stack frame for each method into variables representing the local variables and operand stack slots of the method. Finally, we eliminate the, now unused, *frameStack* from the state of the process, on line 4.

We discuss each of these steps in more detail in separate sections, explaining them with reference to the running example introduced in Section 5.3.1. The removal of launcher returns is discussed first, in Section 5.4.1. Afterwards, the localisation of stack frames is discussed in Section 5.4.2, followed by variable introduction in Section 5.4.3. Finally, we discuss the removal of *frameStack* from the state of the process, in Section 5.4.4.

5.4.1 Remove Launcher Returns

After the previous stage, each conditional branch in a method ends with a return instruction or an infinite loop. This can be seen in Figure 5.14, presented earlier, where the method *TPK_handleAsyncEvent* ends with a *HandleReturnEPC* action. In the first step of this stage, at line 1 of Algorithm 9, such actions are moved outside the method and their definitions are expanded so that their communication with the *Launcher* can be handled. This is performed as described in Algorithm 10, which defines the procedure RemoveLauncherReturns.

Algorithm 10 begins by iterating over each of the method actions, in the for loop beginning on line 1. This determines the name, *methodName*, for each method's action from the class infor-

Algorithm 10 RemoveLauncherReturns

```
1: for $\mathit{methodName} \leftarrow \textsc{MethodActionNames}(\mathit{cs})$ do
2:     $\mathit{methodBody} \leftarrow \textsc{ActionBody}(\mathit{methodName})$
3:     $\mathit{returnAction} \leftarrow \textsc{ReturnAction}(\mathit{methodBody})$
4:     exhaustively apply Law [rec-action-intro]($\mathit{returnAction}$) to $\mathit{methodBody}$
5:     exhaustively apply Rule [conditional-dist]($\mathit{returnAction}$) to $\mathit{methodBody}$
6:     $\textsc{RedefineMethodExcludingReturn}(\mathit{methodName}, \mathit{returnAction})$
7: end for
8: IntroduceFrameStackAssumptions
9: exhaustively apply Rule [refine-$\mathit{HandleReturnEPC}$-empty-$\mathit{frameStack}$]
10: exhaustively apply Rule [refine-$\mathit{HandleReturnEPC}$-nonempty-$\mathit{frameStack}$]
11: exhaustively apply Rule [refine-$\mathit{HandleAreturnEPC}$-empty-$\mathit{frameStack}$]
12: exhaustively apply Rule [refine-$\mathit{HandleAreturnEPC}$-nonempty-$\mathit{frameStack}$]
13: exhaustively apply Law [assump-elim]
14: exhaustively apply Law [seq-unitl]
15: apply Law [copy-rule]($\mathit{ExecuteMethod}$) to $\textsc{ActionBody}(\mathit{MainThread})$
16: apply Law [copy-rule]($\mathit{ExecuteMethod}$) to $\textsc{ActionBody}(\mathit{Started})$
17: apply Rule [$\mathit{ExecuteMethod}$-refinement] to $\textsc{ActionBody}(\mathit{MainThread})$
18: apply Rule [$\mathit{ExecuteMethod}$-refinement] to $\textsc{ActionBody}(\mathit{Started})$
19: apply Law [action-intro]($\mathit{ExecuteMethod}$, $\textsc{ActionBody}(\mathit{ExecuteMethod})$) in reverse
20: match  ($\mathbf{var}\ \mathit{retVal} : \mathit{Word} \bullet (A)(c, m, a, \mathit{retVal});$
               $\mathit{executeMethodRet}!\mathit{thread}!\mathit{retVal} \longrightarrow \mathbf{Skip}$)
21:          in $\textsc{ActionBody}(\mathit{Started})$ then
22:     apply Law [action-intro]($\mathit{ExecuteMethod}$, $A$)
23: apply Law [copy-rule]($\mathit{ExecuteMethod}$) in reverse to $\mathit{MainThread}$
24: apply Law [copy-rule]($\mathit{ExecuteMethod}$) in reverse to $\mathit{Started}$
```

mation, *cs*, via a function MethodActionNames. We take the method's body, *methodBody*, as the body of the action corresponding to *methodName*.

The return actions that may occur at the end of method branches are either *HandleAreturnEPC* or *HandleReturnEPC*. *HandleAreturnEPC* occurs only in methods that return a value and, conversely, *HandleReturnEPC* occurs only in methods that do not return a value. We can thus determine which return action a method uses by examining *methodBody* to see which action occurs at the end of the branches. A method in which all branches end in infinite loops is treated as using the return action *HandleReturnEPC*, since it does not produce a value. We determine the return action type, *returnAction*, for *methodBody* on line 3, using a syntactic function ReturnAction.

With *returnAction* identified, we convert the method to a form in which it has one occurrence of that action at the end of its body. This is achieved by introducing occurrences of *returnAction* after infinite loops in *methodBody* using Law [rec-action-intro] and distributing occurrences of *returnAction* outside conditionals using Rule [conditional-dist], which distributes an action outside a *Circus* conditional and the variable block surrounding it. These laws are applied on lines 4 and 5 of Algorithm 10.

When the method has a single return instruction at the end, it is redefined to exclude the return action. This is performed using Law [copy-rule] and Law [action-intro], but since the use of these laws to redefine an action in this way is standard, it is specified in a separate procedure RedefineMethodExcludingReturn(*methodName*,*returnAction*), called on line 6

Rule [refine-*HandleReturnEPC*-empty-*frameStack*]**.**

$$\begin{array}{l} \{\# \mathit{frameStack} = 1\}; \\ \mathit{HandleReturnEPC} \end{array} \sqsubseteq_A \begin{array}{l} \mathbf{var}\ \mathit{returnValue} : \mathit{Word} \bullet \\ \big(\mathit{InterpreterReturnEPC}\big); \\ \mathit{executeMethodRet}!\mathit{thread}!\mathit{returnValue} \longrightarrow \mathbf{Skip} \end{array}$$

Figure 5.30: Rule [refine-*HandleReturnEPC*-empty-*frameStack*]

of Algorithm 10. This procedure is defined in Algorithm 15, which is included in Appendix A.

We then introduce assumptions that state the depth of the *frameStack*, so that we can determine whether the *frameStack* is empty or not at each return instruction. This introduction of assumptions is performed by the call to the IntroduceFrameStackAssumptions procedure on line 8. It is defined by Algorithm 16, which is included in Appendix A along with the rules used by it.

This procedure introduces an assumption $\{\# \mathit{frameStack} = 0\}$ from the *InterpreterInitEPC* schema (which is the result of applying the data refinement in Section 5.3.7 to *InterpreterInit*) at the start of the main action of $\mathit{ThrCF}_{bc,cs}$. The assumption is then distributed throughout the process by exhaustive application of restricted versions of standard algebraic assumption-distribution laws, and rules stating how the size of *frameStack* is affected by the operations that appear in the code resulting from the elimination of program counter.

The restrictions added to these laws, in the form of extra provisos, guarantee that the assumption is not distributed if an identical assumption is already in place, thus preventing unbounded distribution of the assumptions and ensuring the procedure terminates. The result is that the return instructions following the method actions in *ExecuteMethod* have an assumption $\# \mathit{frameStack} = 1$ before them, and the return instructions occurring in the middle of other methods have an assumption $\# \mathit{frameStack} = k$ for some $k > 1$.

After assumptions on the state of the *frameStack* have been introduced, we can handle the return actions at each point where they occur, applying the rules on lines 9 to 12 of Algorithm 10 wherever possible. An example is Rule [refine-*HandleReturnEPC*-empty-*frameStack*], shown in Figure 5.30. This rule replaces an occurrence of *HandleReturnEPC* where the *frameStack* has a size of 1, with a call to the data operation *InterpreterReturnEPC* followed by a communication with the *Launcher* on the *executeMethodRet* channel. The value communicated on *executeMethodRet* is *returnValue*, introduced in a variable block.

Rule [refine-*HandleReturnEPC*-empty-*frameStack*] essentially expands the definition of the *HandleReturnEPC* action, shown below, and resolves the choice in *CheckLauncherReturn* (presented earlier in Section 4.3.4) over whether *frameStack* is empty. This involves distributing the assumption over the data operation *InterpreterReturnEPC*, which removes the last stack frame from the *frameStack*, causing it to be empty when $\# \mathit{frameStack} = 1$.

$$\begin{array}{l} \mathit{HandleReturnEPC} \mathrel{\widehat{=}} \mathbf{var}\ \mathit{returnValue} : \mathit{Word} \bullet \\ \qquad \big(\mathit{InterpreterReturnEPC}\big) ;\ \mathit{CheckLauncherReturn}(\mathit{returnValue}) \end{array}$$

The other rules used on lines 9 to 12 are similar, handling the cases for the *frameStack* being left nonempty and the *HandleAreturnEPC* action. They can be found in Appendix A.

In the cases when the *frameStack* is not empty after execution of the return instruction, which occurs when a method is called from within another method, the resolution of the

```
$ExecuteMethod \mathrel{\widehat=}$
    $\mathbf{val}\, classID : ClassID;\ \mathbf{val}\, methodID : MethodID;\ \mathbf{val}\, methodArgs : \mathrm{seq}\, Word \bullet$
    $\mathbf{if}(classID, methodID) = (TPKClassID, APEHinit) \longrightarrow$
        $InterpreterNewStackFrame[TPK / class?, APEHinit / methodID?] \mathbin{;}\ Poll;$
        $TPK\_APEHinit;$
        $(\mathbf{var}\, returnValue : Word \bullet \big(InterpreterReturnEPC\big);$
            $executeMethodRet!thread!returnValue \longrightarrow \mathbf{Skip})$
    $[\!]\ (classID, methodID) = (TPKClassID, handleAsyncEvent) \longrightarrow$
        $InterpreterNewStackFrame[TPK / class?, handleAsyncEvent / methodID?] \mathbin{;}\ Poll;$
        $TPK\_handleAsyncEvent;$
        $(\mathbf{var}\, returnValue : Word \bullet \big(InterpreterReturnEPC\big);$
            $executeMethodRet!thread!returnValue \longrightarrow \mathbf{Skip})$
    $[\!]\ (classID, methodID) = (TPKClassID, f) \longrightarrow$
        $InterpreterNewStackFrame[TPK / class?, f / methodID?] \mathbin{;}\ Poll;$
        $TPK\_f;$
        $(\mathbf{var}\, returnValue : Word \bullet \big(InterpreterAreturn2EPC\big);$
            $executeMethodRet!thread!returnValue \longrightarrow \mathbf{Skip})$
    ...
    $\mathbf{fi}$
```

Figure 5.31: *ExecuteMethod* after refining return instructions

choice in *CheckLauncherReturn* collapses it to the branch where there is no communication on *executeMethodRet*. The variable block for *returnValue* is thus removed as part of the rules handling those cases, since it is not used.

After application of these rules, any remaining assumptions are eliminated by application of Law [assump-elim] and Law [seq-unitl] on lines 13 and 14, since they are no longer necessary.

Since the return action at the end of each method in *ExecuteMethod* causes *frameStack* to be empty, the application of these transformations to our running example results in the *ExecuteMethod* action shown in Figure 5.31. Each method action is followed by a *returnValue* variable block with a data operation followed by an *executeMethodRet* communication, which result from the refinement of the return actions. While the data operations differ depending on whether a value is returned from the method or not, the variable block and *executeMethodRet* communication are the same for each method. We thus distribute them outside *ExecuteMethod*, to avoid their unnecessary duplication in each of the branches of *ExecuteMethod*. This is performed by first replacing *ExecuteMethod* with its definition, via an application of Law [copy-rule] on lines 15 and 16, then applying Rule [*ExecuteMethod*-refinement], shown in Figure 5.32, on lines 17 and 18, to distribute the variable block and communication.

After Rule [*ExecuteMethod*-refinement] has been applied, *ExecuteMethod* is redefined as the actions inside the parametrised block on the left-hand side of that rule. This is performed using Law [action-intro] on lines 19 to 22, eliminating the existing definition of *ExecuteMethod* and introducing a new definition. The actions are then copied back out using the new definition by application of Law [copy-rule] on lines 23 and 24. This results in the *MainThread* and *Started* actions shown in Figure 5.33. Note that, since these actions have a very specific format, we can be sure that the application of Law [copy-rule] replaces only the intended components of

Rule [*ExecuteMethod*-refinement]**.** If, for all i, *returnValue* is not free in A_i, then

$$
\begin{array}{lcl}
\begin{array}{l}
(\mathbf{val}\, classID : ClassID;\\
\mathbf{val}\, methodID : MethodID;\\
\mathbf{val}\, methodArgs : \mathrm{seq}\, Word \bullet\\
\mathbf{if}\ (classID, methodID) = (c_1, m_1) \longrightarrow\\
\quad A_1\ ;\ \mathbf{var}\, returnValue : Word \bullet B_1;\\
\quad executeMethodRet!thread!returnValue\\
\quad \longrightarrow \mathbf{Skip}\\
\ldots\\
\ [\!]\ (classID, methodID) = (c_n, m_n) \longrightarrow\\
\quad A_n\ ;\ \mathbf{var}\, returnValue : Word \bullet B_n;\\
\quad executeMethodRet!thread!returnValue\\
\quad \longrightarrow \mathbf{Skip}\\
\mathbf{fi})(c, m, a)
\end{array}
& \sqsubseteq_A &
\begin{array}{l}
\mathbf{var}\, retVal : Word \bullet\\
(\mathbf{val}\, classID : ClassID;\\
\mathbf{val}\, methodID : MethodID;\\
\mathbf{val}\, methodArgs : \mathrm{seq}\, Word;\\
\mathbf{res}\, returnValue : Word \bullet\\
\mathbf{if}\ (classID, methodID) = (c_1, m_1) \longrightarrow\\
\quad A_1\ ;\ B_1\\
\ldots\\
\ [\!]\ (classID, methodID) = (c_n, m_n) \longrightarrow\\
\quad A_n\ ;\ B_n\\
\mathbf{fi})(c, m, a, retVal);\\
executeMethodRet!thread!retVal\\
\ \longrightarrow \mathbf{Skip}
\end{array}
\end{array}
$$

Figure 5.32: Rule [*ExecuteMethod*-refinement]

$$
\begin{array}{l}
MainThread \mathrel{\widehat=}\\
\quad setStack?t : (t = thread)?stack \longrightarrow frameStackID := Initialised\ stack\ ;\ \mu X \bullet\\
\quad \left(\begin{array}{l}
\mathbf{var}\, retVal : Word \bullet executeMethod?t : (t = thread)?c?m?a \longrightarrow\\
\quad ExecuteMethod(c, m, a, retVal)\ ;\ executeMethodRet!thread!retVal \longrightarrow Poll\ ;\ X\\
\Box\\
CEEswitchThread?from?to : (from = thread) \longrightarrow Blocked\ ;\ X
\end{array}\right)\\
Started \mathrel{\widehat=}\\
\quad \left(\begin{array}{l}
\mathbf{var}\, retVal : Word \bullet executeMethod?t : (t = thread)?c?m?a \longrightarrow;\\
\quad ExecuteMethod(c, m, a, retVal)\ ;\ executeMethodRet!thread!retVal \longrightarrow Poll;\\
\quad \left(\begin{array}{l}
continue?t : (t = thread) \longrightarrow Started\\
\Box\\
endThread?t : (t = thread) \longrightarrow \mathbf{Skip}
\end{array}\right)\\
\Box\\
CEEswitchThread?from?to : (from = thread) \longrightarrow Blocked\ ;\ Started\\
\Box\\
endThread?t : (t = thread) \longrightarrow \mathbf{Skip}
\end{array}\right);\\
\quad removeThreadMemory!thread \longrightarrow CEEremoveThread!thread\\
\quad \longrightarrow CEEswitchThread?from?to : (from = thread) \longrightarrow NotStarted
\end{array}
$$

Figure 5.33: *MainThread* and *Started* after *Launcher* return elimination

those actions. Within the body of a method, the return actions have been refined to simple data operations with no *executeMethodRet* communication, as can be seen in Figure 5.34, which shows the form of *TPK_handleAsyncEvent*.

5.4.2 Localise Stack Frames

After the *CheckLauncherReturn* actions have been handled, the process no longer has any actions that use the whole *frameStack*. We can therefore refine each method to only operate on a local stack frame variable. This is performed as described in Algorithm 11, which defines the procedure LOCALISESTACKFRAMES.

$$
\begin{array}{l}
\mathit{TPK\_handleAsyncEvent} \mathrel{\widehat=} \\
\quad \mathit{HandleNewEPC}(27) \mathbin{;} \mathit{Poll} \mathbin{;} \mathit{HandleDupEPC} \mathbin{;} \mathit{Poll} \mathbin{;} \mathit{HandleAconst\_nullEPC} \mathbin{;} \mathit{Poll}; \\
\quad (\mathbf{var}\, \mathit{poppedArgs} : \operatorname{seq} \mathit{Word} \bullet \\
\qquad \big(\exists\, \mathit{argsToPop?} == 2 \bullet \mathit{InterpreterStackFrameInvoke}\big); \\
\qquad \big(\mathit{InterpreterNewStackFrame}[\\
\qquad\quad \mathit{ConsoleConnection}/\mathit{class?}, \mathit{CCinit}/\mathit{methodID?}, \mathit{poppedArgs}/\mathit{methodArgs?}]\big)) \mathbin{;} \mathit{Poll}; \\
\quad \mathit{ConsoleConnection\_CCinit} \mathbin{;} \big(\mathit{InterpreterReturnEPC}\big) \mathbin{;} \mathit{Poll}; \\
\quad \ldots \\
\qquad\quad (\mathbf{var}\, \mathit{poppedArgs} : \operatorname{seq} \mathit{Word} \bullet \\
\qquad\qquad \big(\exists\, \mathit{argsToPop?} == 1 \bullet \mathit{InterpreterStackFrameInvoke}\big); \\
\qquad\qquad \big(\mathit{InterpreterNewStackFrame}[\\
\qquad\qquad\quad \mathit{TPK}/\mathit{class?}, f/\mathit{methodID?}, \mathit{poppedArgs}/\mathit{methodArgs?}]\big)); \\
\qquad\quad \mathit{Poll} \mathbin{;} \mathit{TPK\_f} \mathbin{;} \big(\mathit{InterpreterAreturn1EPC}\big) \mathbin{;} \mathit{Poll}; \\
\qquad\quad \ldots \\
\qquad\quad \mathit{Poll} \mathbin{;} \mathit{HandleAstoreEPC}(4) \mathbin{;} \mathit{Poll} \mathbin{;} Y \\
\qquad [\!] \ \mathit{value}1 > \mathit{value}2 \longrightarrow \mathbf{Skip} \\
\qquad \mathbf{fi} \mathbin{;} \mathit{Poll}
\end{array}
$$

Figure 5.34: *TPK_handleAsyncEvent* after *Launcher* return elimination

Algorithm 11 LocaliseStackFrames

```
1: apply Law [forwards-data-refinement]($\mathit{InterpreterStateFS}$, $\mathit{FrameStackCI}$)
2: $\mathit{iterationOrder} \leftarrow$ MethodDependencyOrder
3: for $\mathit{methodName} \leftarrow \mathit{iterationOrder}$ do
4:     $\mathit{numArgs} \leftarrow$ MethodArguments($\mathit{methodName}$)
5:     if $\neg$ IsStatic($\mathit{methodName}$) then
6:         $\mathit{numArgs} \leftarrow \mathit{numArgs} + 1$
7:     end if
8:     exhaustively apply Rule [$\mathit{InterpreterReturn}$-args-intro]($\mathit{methodName}$, $\mathit{numArgs}$)
9:     RedefineMethodToIncludeParameters($\mathit{methodName}$)
10:    apply Rule [$\mathit{InterpreterReturn}$-$\mathit{stackFrame}$-intro]($\mathit{numArgs}$)
           to ActionBody($\mathit{methodName}$)
11: end for
```

We first apply a data refinement to remove *currentClass* from the state. We have defined *currentClass* in the model as a convenience when accessing the *frameClass* of the topmost stack frame, which is no longer necessary when we have separate variables for each stack frame. The data refinement is applied on line 1 of Algorithm 11, and transforms the state to *InterpreterStateFS*, below, which only contains *frameStack*.

$$
\begin{array}{|l}
\hline
\mathit{InterpreterStateFS} \\
\quad \mathit{frameStack} : \operatorname{seq} \mathit{StackFrameEPC} \\
\hline
\end{array}
$$

The relationship between *InterpreterStateEPC* and *InterpreterStateFS* is described by the coupling invariant *FrameStackCI*, shown below. It ensures *frameStack* is unaffected by the refinement and replaces occurrences of *currentClass* with (*last frameStack*).*frameClass*.

$$
\begin{array}{|l}
\textit{FrameStackCI} \\
\hline
\textit{InterpreterStateEPC} \\
\textit{InterpreterStateFS}_1 \\
\hline
\textit{frameStack} = \textit{frameStack}_1 \\
\textit{currentClass} = (\textit{last}\,\textit{frameStack}_1).\textit{frameClass} \\
\hline
\end{array}
$$

FrameStackCI describes a functional data refinement, so the new actions can be calculated in each case. The effect of this data refinement is, as mentioned above, that *currentClass* is replaced with (*last frameStack*).*frameClass*, wherever it occurs in the old actions. We can then proceed with introducing stack frame variables.

When introducing these variables, we must begin with those methods at the greatest call depth. This is necessary due to the approach we take in introducing these variables, as we explain later in this section. We therefore introduce stack frame variables to the methods in the order specified by a procedure MethodDependencyOrder, which constructs a sequence of method action names indicating the order in which the method actions should be handled. This sequence is constructed by first adding to the sequence any methods that contain no method calls, then adding any methods that only call methods already in the sequence, and repeating until all methods are in the sequence. Since we do not allow recursion, this always terminates. We construct this sequence and assign it to *iterationOrder* on line 2 of Algorithm 11.

We then loop, introducing a stack frame variable for each method in the order specified by *iterationOrder*, in the for loop on line 3. Within the for loop, we first introduce value parameters, representing the arguments to the method, around the call to the method action. This ensures that the body of the method is completely independent of the context in which it is called, enabling us to separate the whole method body (including stack frame creation and return actions) into its own action. Introduction of method arguments is performed using Rule [*InterpreterReturn*-args-intro], shown in Figure 5.35. This rule is applied to two parameters: *methodName*, the name of the method being considered, and *numArgs*, the number of arguments to the method. The number of arguments is determined from the name of the method (since the arguments of a method are encoded in its identifier) and we add an extra argument for `this` if the method is not static, as indicated by the if statement on lines 5 to 7. We apply this rule everywhere it applies on line 10.

Rule [*InterpreterReturn*-args-intro]**.** Given an action name M and $n : \mathbb{N}$, if $arg1, \ldots, arg{<}n{>}$ are not free in M, are distinct from c, m and $args$, and $\# \, args = n$, then

$$
\begin{array}{l}
\big(\mathit{InterpreterNewStackFrame}[\\
\quad c / \mathit{class}?, \\
\quad m / \mathit{methodID}?, \\
\quad \mathit{args} / \mathit{methodArgs}?]\big); \\
\mathit{Poll} \; ; \; M \; ; \; \big(\mathit{InterpreterReturn}\big)
\end{array}
\sqsubseteq_A
\begin{array}{l}
(\mathbf{val}\; \mathit{arg}1, \ldots, \mathit{arg}{<}n{>} : \mathit{Word} \bullet \\
\quad \big(\exists\, \mathit{methodArgs}? == \langle \mathit{arg}1, \ldots, \mathit{arg}{<}n{>} \rangle \bullet \\
\qquad \mathit{InterpreterNewStackFrame}[\\
\qquad\quad c / \mathit{class}?, \\
\qquad\quad m / \mathit{methodID}?]\big); \\
\quad \mathit{Poll} \; ; \; M \; ; \; \big(\mathit{InterpreterReturn}\big) \\
)(\mathit{args}\, 1, \ldots, \mathit{args}\; n)
\end{array}
$$

Figure 5.35: Rule [*InterpreterReturn*-args-intro]

Rule [*InterpreterReturn*-args-intro] introduces value parameters representing method arguments around a method ending with an *InterpreterReturn* operation. The arguments array passed as the *methodArgs* input of *InterpreterNewStackFrame* is split into its individual elements, which are passed to the parameters and then recombined to be passed into *InterpreterNewStackFrame*. This splitting of the array ensures that the individual arguments can be more easily handled in the next step, where we introduce local variables.

For example, every call to the method action *TPK_handleAsyncEvent* is preceded by an *InterpreterNewStackFrame* operation and followed by an *InterpreterReturnEPC* operation. The Rule [*InterpreterReturn*-args-intro] therefore applies to it and the number of arguments given as the n parameter to the rule is 1, since the SCJ method `handleAsyncEvent()` takes no explicit arguments, but is not static. Calls to *TPK_handleAsyncEvent* (the only one of which occurs in *ExecuteMethod*, since it is only called by the infrastructure), then have the form shown below.

$$
\begin{array}{l}
(\mathbf{val}\; arg1 : Word \bullet \\
\quad \big(\exists\, methodArgs? == \langle arg1 \rangle \bullet \\
\quad\quad InterpreterNewStackFrame[TPK / class?, handleAsyncEvent / methodID?]\big); \\
\quad\quad Poll\; ;\; TPK\_handleAsyncEvent\; ;\; \big(InterpreterReturn\big) \\
)(methodArgs\, 1)
\end{array}
$$

After the method arguments have been introduced, we redefine the method action to include the contents of the parametrised block introduced by Rule [*InterpreterReturn*-args-intro]. This is performed in a separate procedure, RedefineMethodToIncludeParameters, which is similar to the RedefineMethodExcludingReturn procedure used in the previous section. It is defined by Algorithm 17 in Appendix A.

Having completely separated the method into its own, independent, action, we then introduce the stack frame variable for the method using Rule [*InterpreterReturn-stackFrame*-intro], shown in Figure 5.36. This is applied to the body of *methodName* on line 10, with the number of arguments, *numArgs*, passed to it.

Rule [*InterpreterReturn-stackFrame*-intro] introduces a variable *stackFrame*, which has type *StackFrameEPC*, over the body of a method that ends with an *InterpreterReturn* operation. The *stackFrame* variable is initialised, in *Init<c>_<m>SF*, in the same way as for the stack frame created by *InterpreterNewStackFrame*, and each reference to *last frameStack* in the body of the method is replaced with a reference to *stackFrame*. Replacing the references to *last frameStack* requires that the size of *frameStack* does not change during the method. However, this requirement is met since method calls are the only operations that change the size of *frameStack* and we replace references to the *frameStack* in nested methods first, by the definition of *iterationOrder*.

Iterating over methods beginning with the greatest call depth ensures that the requirements of Rule [*InterpreterReturn-stackFrame*-intro] are met. Otherwise, each nested method call would have its own *InterpreterNewStackFrame* operation, as can be seen in Figure 5.34, where the call to *TPK_f* has such an operation. This changes the size of *frameStack*, making references to *last frameStack* refer to a different stack frame and so preventing a direct replacement with the *stackFrame* variable. Ensuring the stack frame variable is introduced for *TPK_f* first avoids this issue, since it means that references to *frameStack* (including the operation to change its size) are already replaced with references to a separate *stackFrame* variable by the time we apply Rule [*InterpreterReturn-stackFrame*-intro] to *TPK_handleAsyncEvent*.

Note that the operations performed on lines 8 and 10 of Algorithm 11 specifically handle methods that do not return a value. We omit the similar handling of methods that do return a value. Handling such methods requires rules similar to Rule [*InterpreterReturn*-args-intro] for

Rule [*InterpreterReturn-stackFrame*-intro]**.** Given $n : \mathbb{N}$, if the only occurrences of *frameStack* in A are in the expression *last frameStack*, the length of *frameStack* does not change throughout A, and *stackFrame* is not free in A, then

$$\begin{array}{l}\big(\exists\, methodArgs? == \langle arg1, \ldots, arg{<}n{>}\rangle \bullet \\ \quad InterpreterNewStackFrame[\\ \qquad c/class?, \\ \qquad m/methodID?]\big); \\ A\ ;\ \big(InterpreterReturn\big)\end{array} \quad \sqsubseteq_A \quad \begin{array}{l}\mathbf{var}\ stackFrame : StackFrameEPC \bullet \\ \quad \big(Init{<}c{>}\_{<}m{>}SF\big); \\ \quad A[stackFrame/last\ frameStack, \\ \qquad stackFrame'/last\ frameStack']\end{array}$$

where *Init<c>_<m>SF* is defined by

$$\begin{array}{|l}\hline Init{<}c{>}\_{<}m{>}SF \\ \quad arg1?, \ldots, arg{<}n{>}? : Word \\ \quad stackFrame' : StackFrameEPC \\ \hline \quad \langle arg1?, \ldots, arg{<}n{>}?\rangle \text{ prefix } stackFrame'.localVariables \\ \quad \#\, stackFrame'.localVariables = c.methodLocals\ m \\ \quad stackFrame'.operandStack = \langle\rangle \\ \quad stackFrame'.frameClass = c \\ \quad stackFrame'.stackSize = c.methodStackSize\ m \\ \hline\end{array}$$

Figure 5.36: Rule [*InterpreterReturn-stackFrame*-intro]

method bodies followed by *InterpreterAreturn*1 and *InterpreterAreturn*2, which introduce a result parameter for the method in addition to the value parameters representing the method's arguments. The new method action then has to match the different method parameters. We also require a rule similar to Rule [*InterpreterReturn-stackFrame*-intro] to handle the slightly different ending of the method action that the return handling creates. These rules are applied in a way similar to the existing rules.

In our example, the body of *TPK_handleAsyncEvent* (having the form of the actions in the parametrised block shown above), begins with an *InterpreterNewStackFrame* operation having *TPK* as its *class*? and *handleAsyncEvent* as its *methodID*?. This operation is thus replaced with an *InitTPK_handleAsyncEventSF* operation, of the form shown in Rule [*InterpreterReturn-stackFrame*-intro]. After this rule has been applied, *TPK_handleAsyncEvent* is as shown in Figure 5.37.

For brevity, we define new actions, which we refer to as *Handle∗SF* actions. These are not formally introduced as actions in the compilation strategy as they are an abbreviation used for presenting examples and stating compilation rules. They are refined to a different form later in the elimination of frame stack stage. The *Handle∗SF* actions are similar to the *Handle∗EPC* actions, except they have every reference to *last frameStack* (or *last frameStack*′) replaced with a reference to *stackFrame* (or *stackFrame*′), and have undergone the data refinement described above. We name them by replacing *EPC* in the names of the *Handle∗EPC* actions with *SF*. Similarly, we define an *InvokeSF* operation that performs the operation of *InterpreterStackFrameInvoke* over *stackFrame* instead of *last frameStack*.

```
$TPK\_f \mathrel{\widehat=}$
  $\mathbf{val}\ arg1 : Word \bullet$
  $\mathbf{var}\ stackFrame : StackFrameEPC \bullet$
  $\big(InitTPK\_handleAsyncEventSF\big);$
  $Poll \mathbin{;} HandleNewSF(27) \mathbin{;} Poll \mathbin{;} HandleDupSF \mathbin{;} Poll \mathbin{;} HandleAconst\_nullSF;$
  $Poll \mathbin{;} (\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\ argsToPop? == 2 \bullet InvokeSF\big);$
  $ConsoleConnection\_CCinit(poppedArgs\ 1, poppedArgs\ 2)) \mathbin{;} Poll;$
  $HandleAstoreSF(1) \mathbin{;} Poll \mathbin{;} HandleAloadSF(1) \mathbin{;} Poll \mathbin{;} (\mathbf{var}\ poppedArgs : Word \bullet$
  $\big(\exists\ argsToPop? == 1 \bullet InvokeSF\big) \mathbin{;} getClassIDOf!head\ poppedArgs?cid \longrightarrow$
  $\mathbf{if}\ cid = ConsoleConnectionID \longrightarrow ConsoleConnection\_openInputStream(poppedArgs\ 1)$
  $\mathbf{fi}) \mathbin{;} Poll \mathbin{;} HandleAstoreSF(2) \mathbin{;} Poll \mathbin{;} HandleAloadSF(1) \mathbin{;} Poll;$
  $(\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\ argsToPop? == 1 \bullet InvokeSF\big);$
  $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleConnectionID \longrightarrow$
    $ConsoleConnection\_openOutputStream(poppedArgs\ 1)$
  $\mathbf{fi}) \mathbin{;} Poll \mathbin{;} HandleAstoreSF(3) \mathbin{;} Poll \mathbin{;} HandleIconstSF(0) \mathbin{;} Poll;$
  $HandleAstoreSF(4) \mathbin{;} Poll \mathbin{;} Poll \mathbin{;} (\mu\, Y \bullet$
    $HandleAloadSF(4) \mathbin{;} Poll \mathbin{;} HandleIconstSF(10) \mathbin{;} Poll \mathbin{;} (\mathbf{var}\ value1, value2 : Word \bullet$
    $\big(InterpreterPopSF[value2!/value!]\big) \mathbin{;} \big(InterpreterPopSF[value1!/value!]\big)$
    $\mathbf{if}\ value1 \leq value2 \longrightarrow Poll \mathbin{;} HandleAloadSF(2) \mathbin{;} Poll;$
      $(\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \big(\exists\ argsToPop? == 1 \bullet InvokeSF\big);$
      $getClassIDOf!(head\ poppedArgs)?cid \longrightarrow \mathbf{if}\ cid = ConsoleInputClassID \longrightarrow$
        $ConsoleInput\_read(poppedArgs\ 1)$
      $\mathbf{fi}) \mathbin{;} Poll \mathbin{;} (\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet \big(\exists\ argsToPop? == 1 \bullet InvokeSF\big);$
      $TPK\_f(poppedArgs\ 1));$
      $Poll \mathbin{;} HandleAstoreSF(5) \mathbin{;} Poll \mathbin{;} HandleAloadSF(5) \mathbin{;} HandleIconstSF(400);$
      $Poll \mathbin{;} (\mathbf{var}\ value1, value2 : Word \bullet \big(InterpreterPopSF[value2!/value!]\big);$
      $\big(InterpreterPopSF[value1!/value!]\big);$
      $\mathbf{if}\ value1 \leq value2 \longrightarrow HandleAloadSF(3) \mathbin{;} Poll \mathbin{;} HandleAloadSF(5);$
        $Poll \mathbin{;} (\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\ argsToPop? == 2 \bullet InvokeSF\big);$
        $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
          $ConsoleOutput\_write(poppedArgs\ 1, poppedArgs\ 2)$
        $\mathbf{fi})$
      $[]\ value1 > value2 \longrightarrow HandleAloadSF(3) \mathbin{;} Poll \mathbin{;} HandleIconstSF(0);$
        $Poll \mathbin{;} (\mathbf{var}\ poppedArgs : Word \bullet \big(\exists\ argsToPop? == 2 \bullet InvokeSF\big);$
        $getClassIDOf!head\ poppedArgs?cid \longrightarrow \mathbf{if}\ cid = ConsoleOutputID \longrightarrow$
          $ConsoleOutput\_write(poppedArgs\ 1, poppedArgs\ 2)$
        $\mathbf{fi}) \mathbin{;} Poll$
      $\mathbf{fi}) \mathbin{;} Poll \mathbin{;} HandleAloadSF(4) \mathbin{;} Poll \mathbin{;} HandleIconstSF(1) \mathbin{;} Poll;$
      $HandleIaddSF \mathbin{;} Poll \mathbin{;} HandleAstoreSF(4) \mathbin{;} Poll \mathbin{;} Y$
    $[]\ value1 > value2 \longrightarrow \mathbf{Skip}$
    $\mathbf{fi})) \mathbin{;} Poll$
```

Figure 5.37: *TPK_handleAsyncEvent* after its *stackFrame* variable is introduced

Algorithm 12 IntroduceVariables

```
 1: for $methodName \leftarrow$ MethodNames($cs$) do
 2:     IntroduceFrameClassAssumptions(ActionBody($methodName$))
 3:     exhaustively apply Rule [refine-$PutfieldSF$] to ActionBody($methodName$)
 4:     exhaustively apply Rule [refine-$GetfieldSF$] to ActionBody($methodName$)
 5:     exhaustively apply Rule [refine-$PutstaticSF$] to ActionBody($methodName$)
 6:     exhaustively apply Rule [refine-$GetstaticSF$] to ActionBody($methodName$)
 7:     exhaustively apply Rule [refine-$NewSF$] to ActionBody($methodName$)
 8:     exhaustively apply Law [assump-elim] to ActionBody($methodName$)
 9:     exhaustively apply Law [seq-unitl] to ActionBody($methodName$)
10:     IntroduceOperandStackAssumptions(ActionBody($methodName$))
11:     $\ell \leftarrow$ MethodLocals($methodName$)
12:     $s \leftarrow$ MethodStackSize($methodName$)
13:     apply Law [forwards-data-refinement] ($[var1, \ldots, var{<}\ell{>} : Word;$   $, V{<}\ell{>}S{<}s{>}CI$)
                                                  $stack1, \ldots, stack{<}s{>} : Word]$
         to ActionBody($methodName$)
14:     exhaustively apply Law [assump-elim] to ActionBody($methodName$)
15:     exhaustively apply Law [seq-unitl] to ActionBody($methodName$)
16:     exhaustively apply Rule [cond-$value$1-$value$2-elim] to ActionBody($methodName$)
17:     exhaustively apply Rule [$getField$-$oid$-elim] to ActionBody($methodName$)
18:     exhaustively apply Rule [$putField$-$oid$-$value$-elim] to ActionBody($methodName$)
19:     exhaustively apply Rule [$putStatic$-$value$-elim] to ActionBody($methodName$)
20:     exhaustively apply Rule [$poppedArgs$-elim]
21:     exhaustively apply Rule [$poppedArgs$-sync-elim]
22:     exhaustively apply Rule [invokevirtual-$poppedArgs$-elim]
23:     match  $\mu X \bullet A$ ;  if $b \longrightarrow B$ ; $X$     in ActionBody($methodName$) then
                                  $\,[\!]\, \neg\, b \longrightarrow$ Skip
                                  fi
24:         exhaustively apply Law [rec-rolling-rule](($\lambda X \bullet A$ ; $X$), ($\lambda X \bullet$ if $b \longrightarrow B$ ; $X$ ))
                                                                              $\,[\!]\, \neg\, b \longrightarrow$ Skip
                                                                              fi
25:     apply Rule [$var$-parameter-conversion] to ActionBody($methodName$)
26:     RedefineMethodActionToExcludeParameters($methodName$)
27:     apply Rule [argument-variable-elimination]($methodName$)
28: end for
```

5.4.3 Introduce Variables

Following the introduction of local *stackFrame* variables, we perform local data refinements to introduce the local variables and stack slots for each *stackFrame*, on line 3 of Algorithm 9. This is performed as described in Algorithm 12, which defines the IntroduceVariables procedure.

Algorithm 12 operates upon each of the method actions in turn on line 1, determining the names of the actions from *cs* via the function MethodNames. Within this loop we refer to the name of the method action under consideration as *methodName*. Unlike the introduction of the *stackFrame* variables, the order in which the methods are iterated over does not matter, since each has its own *stackFrame* variable that undergoes local refinement.

We first refine field access operations to remove their reliance on the *frameClass* component

Rule [refine-*PutfieldSF*].

$$
\begin{array}{l}
\{stackFrame.frameClass = c\};\\
PutfieldSF(cpi)
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
(\mathbf{var}\ oid : ObjectID;\ value : Word \bullet\\
\quad \big(InterpreterPop[\\
\qquad stackFrame / last\ frameStack,\\
\qquad stackFrame' / last\ frameStack']\big);\\
\quad \big(InterpreterPop[\\
\qquad oid! / value!,\\
\qquad stackFrame / last\ frameStack,\\
\qquad stackFrame' / last\ frameStack']\big);\\
\quad putField!oid!cid!fid!value \longrightarrow \mathbf{Skip})
\end{array}
$$

where

$$
\begin{array}{l}
cpi \in fieldRefIndices\ c \land\\
c.constantPool\ cpi = FieldRef\ (cid, fid)
\end{array}
$$

Figure 5.38: Rule [refine-*PutfieldSF*]

of the *stackFrame*, which is removed in the data refinement later in this section. This is done by first introducing and distributing assumptions stating the value of *frameClass*, using the procedure IntroduceFrameClassAssumptions on line 2, which is applied to the body of *methodName*. This procedure is similar to IntroduceFrameStackAssumptions in that it introduces an assumption and distributes it with restricted forms of standard algebraic laws. However, IntroduceFrameClassAssumptions acts on the body of a single method and introduces an assumption about the value of the *frameClass* component of *stackFrame* from the schema action initialising *stackFrame*. The IntroduceFrameClassAssumptions procedure is defined by Algorithm 18, which we omit here, but is included in Appendix A.

We can then apply Rules [refine-*PutfieldSF*], [refine-*GetfieldSF*], [refine-*PutstaticSF*], [refine-*GetstaticSF*] and [refine-*NewSF*] wherever possible to refine the field accesses and object creation actions, on lines 3 to 7. As an example of one of these rules, we show Rule [refine-*PutfieldSF*] in Figure 5.38. It refines a *PutfieldSF*(*cpi*) instruction preceded by an assumption stating the value of the *frameClass* component of *stackFrame*. With the application of the rule, the definition of *PutfieldSF* is expanded and the class identifier, *cid*, and field identifier, *fid*, at the constant pool index *cpi* are substituted in place of the accesses to the constant pool. This removes the reference to the *constantPool* of the *frameClass*, and hence the reference to the *frameClass*. Rule [refine-*GetfieldSF*], Rule [refine-*PutstaticSF*], Rule [refine-*GetstaticSF*] and Rule [refine-*NewSF*] are similar so we omit them here. They can be found, along with the other compilation rules, in Appendix A. After these laws have been applied, we eliminate any remaining *frameClass* assumptions by applying Law [assump-elim] and Law [seq-unitl] on lines 8 and 9.

After references to the *frameClass* have been removed, we can perform a local data refinement on the body of the method to convert the *stackFrame* variable to separate variables for the local variables and operand stack slots. Since we are converting the *operandStack* component of *stackFrame* from a sequence to a fixed set of variables representing an array of stack slots, we must know the length of *operandStack* before each operation in order to determine which variable corresponds to the top of the stack, and hence should be affected by the operation.

We ensure that this information is available, by introducing and distributing assumptions on the size of *operandStack*. This is performed by the IntroduceOperandStackAssumptions procedure, called on line 10. It is similar to IntroduceFrameClassAssumptions and is defined by Algorithm 19, which is included in Appendix A.

After an *operandStack* assumption has been introduced before each data operation, the local data refinement is performed on line 13. The new state for the data refinement contains the local variables, which are all of type *Word*, and are named *var* followed by an integer beginning at 1 and going up to the total number of local variables for the method, ℓ. It also contains operand stack slots, named *stack* followed by an integer from 1 up to the maximum stack size for the method, s. These values ℓ and s are obtained from the *methodLocals* and *methodStackSize* information for the method in cs, on lines 11 and 12 of Algorithm 12. For example, the values associated with *handleAsyncEvent* in the *TPK* class in Figure 5.5 are 6 for ℓ and 3 for s. The coupling invariant for the data refinement of a method m is then given by the template $V{<}\ell{>}S{<}s{>}CI$, shown below.

$$
\begin{array}{|l}
\hline
V{<}\ell{>}S{<}s{>}CI \\
\hline
stackFrame : StackFrameEPC \\
var1, \ldots, var{<}\ell{>} : Word \\
stack1, \ldots, stack{<}s{>} : Word \\
\hline
\# stackFrame.localVariables = {<}\ell{>} \\
stackFrame.localVariables\ 1 = var1 \\
\qquad \vdots \\
stackFrame.localVariables\ {<}\ell{>} = var{<}\ell{>} \\
stackFrame.stackSize = {<}s{>} \\
\# stackFrame.operandStack \geq 1 \Rightarrow \\
\qquad stackFrame.operandStack\ 1 = stack1 \\
\qquad \vdots \\
\# stackFrame.operandStack \geq {<}s{>} \Rightarrow \\
\qquad stackFrame.operandStack\ {<}s{>} = stack{<}s{>} \\
\hline
\end{array}
$$

$V{<}\ell{>}S{<}s{>}CI$ requires the number of local variables to be equal to ℓ, and relates each of the values in the *localVariables* sequence in *stackFrame* to the corresponding local variables, *var*1 to *var*<ℓ>. It also requires the maximum operand stack size, *stackSize*, to be s, and relates each value in *operandStack* to the corresponding stack slots, *stack*1 to *stack*<s>, but only if *operandStack* is long enough to contain such a value. The values of the stack slots outside the length of the *operandStack* at each point in the program are not specified, and so are chosen nondeterministically, since they are not used until they have been initialised with the correct value. This nondeterminism allows us to avoid introducing unnecessary assignments to initialise the stack slots and return them to a default value when they are no longer used, which would be required if we specified a value for unused stack slots in the coupling invariant.

However, the nondeterminism in $V{<}\ell{>}S{<}s{>}CI$ means that it does not define a function, since there are multiple possible states for the operand stack slots that correspond to a non-full *operandStack*. This means that we cannot directly compute the actions resulting from the refinement (since there are multiple possibilities), and must specify how each of the data operations is refined. We, therefore, state 12 compilation rules in terms of *Circus* simulations between actions.

Most of the bytecode instructions at this stage in the strategy have their semantics stated in terms of a data operation over *stackFrame*, in the form of a *Handle∗SF* action. We state the simulations for such instructions as simulations of a *Handle∗SF* action preceded by an *operandStack* size assumption, which can be viewed as adding an extra precondition to the action. An example of such a simulation is Rule [*HandleAloadSF*-simulation], shown in Figure 5.39. It states that *HandleAloadSF*(*lvi*), with an assumption that the size of *operandStack* is k, is simulated by an assignment $stack{<}k + 1{>} := var{<}lvi + 1{>}$. Note that the local variable index *lvi* has 1 added to it, since Java's indices start at 0, whereas Z sequences, and hence our variable numbering, are indexed starting at 1. This rule applies, for example, to the *HandleAloadSF*(1) action deriving from the *aload* 1 instruction at $pc = 12$ in *TPK_handleAsyncEvent*, which is refined to $stack1 := var2$, since the stack is empty at that point.

Rule [*HandleAloadSF*-simulation]**.**

$$\begin{array}{l} \{\#\, stackFrame.operandStack = k\}; \\ HandleAloadSF(lvi) \end{array} \quad \preccurlyeq \quad stack{<}k + 1{>} := var{<}lvi + 1{>}$$

Figure 5.39: Rule [*HandleAloadSF*-simulation]

The instructions that manipulate objects by communicating with the object manager have already had their definitions expanded earlier in this stage. Their communication with the object manager need not be changed by the data refinement. However, the data operations used by these operations to pop or push the values communicated from or to the operand stack must be refined. An example of the simulation for such an operation is Rule [*InterpreterPopEPC*-simulation], shown in Figure 5.40. This establishes a simulation between *InterpreterPopEPC*, modified to act over *stackFrame*, and an assignment of a stack slot value to the variable *value*, which is in scope in the contexts where *InterpreterPopEPC* is used.

Rule [*InterpreterPopEPC*-simulation]**.**

$$\begin{array}{l} \{\#\, stackFrame.operandStack = k\}; \\ \big(InterpreterPopEPC[\\ \quad stackFrame / last\, frameStack, \\ \quad stackFrame' / last\, frameStack']\big) \end{array} \quad \preccurlyeq \quad value := stack{<}k{>}$$

Figure 5.40: Rule [*InterpreterPopEPC*-simulation]

Method invocations also use data operations to pop the method's arguments from the stack and pass them to the method, which must be refined in the data refinement. The *InvokeSF* operation, which pops the method's arguments from the stack, is simulated by an assignment of a sequence of operand stack values to the *poppedArgs* variables, as stated in Rule [*InvokeSF*-simulation], shown in Figure 5.41.

The passing of the arguments to the invoked method has already been refined in the introduction of the *stackFrame* variable, but the schema initialising *stackFrame* must be further refined to initialise the local variables. The simulation for the *stackFrame* initialisation schema is stated by Rule [*stackFrame*-init-simulation], shown in Figure 5.42. It is simulated by a sequence of assignments setting the local variables to the values of the arguments. The initialisation sets *operandStack* to be empty, so there is no need to assign values to the stack slot variables; they can be left arbitrary.

Rule [*InvokeSF*-simulation].

$$\begin{array}{l}\{\#\, \mathit{stackFrame}.\mathit{operandStack} = k\};\\ \big(\exists\, \mathit{argsToPop}? == m \bullet \mathit{InvokeSF}\big)\end{array} \preccurlyeq \begin{array}{l}\mathit{poppedArgs} :=\\ \qquad \langle \mathit{stack}{<}k - m + 1{>}, \ldots, \mathit{stack}{<}k{>}\rangle\end{array}$$

Figure 5.41: Rule [*InvokeSF*-simulation]

Rule [*stackFrame*-init-simulation].

$$\begin{array}{l}\big([\mathit{arg}1?, \ldots, \mathit{arg}{<}n{>}? : \mathit{Word};\\ \quad \mathit{stackFrame}' : \mathit{StackFrameEPC} \mid\\ \quad \langle \mathit{arg}_1, \ldots, \mathit{arg}_n\rangle \subseteq \mathit{stackFrame}'.\mathit{localVariables} \land\\ \quad \#\, \mathit{stackFrame}'.\mathit{localVariables} = \ell \land\\ \quad \mathit{stackFrame}'.\mathit{operandStack} = \langle\rangle \land\\ \quad \mathit{stackFrame}'.\mathit{frameClass} = c \land\\ \quad \mathit{stackFrame}'.\mathit{stackSize} = s]\big)\end{array} \preccurlyeq \begin{array}{l}\mathit{var}{<}1{>} := \mathit{arg}_1;\\ \qquad \vdots\\ \mathit{var}{<}n{>} := \mathit{arg}_n\end{array}$$

Figure 5.42: Rule [*stackFrame*-init-simulation]

The simulation rules we have omitted here can be found with the rest of the compilation rules in Appendix A. These, together with the standard laws for distributing simulations through *Circus* constructs, are sufficient to unambiguously define the local data refinement to be applied to the method. After the data refinement, we eliminate any remaining assumptions by applying Law [assump-elim] and Law [seq-unitl] on lines 14 and 15.

We then eliminate the additional variables used in the data operations that pop values from the stack. Those that push values to the stack are pushing values received from a channel, which require a separate assignment operation and so cannot be eliminated. The additional variables are eliminated using the rules applied on lines 16 to 19 of Algorithm 12. In particular, Rule [cond-*value*1-*value*2-elim], shown in Figure 5.43, applies to the *TPK_handleAsyncEvent* action in our example. It removes the need for additional *value*1 and *value*2 variables, replacing the references to them in the conditional with the stack slot variables whose values they store.

We also eliminate the intermediate *poppedArgs* variable used when passing variables to method calls in the body of a method. This is performed by the rules applied on lines 20 to 22, which are

Rule [cond-*value*1-*value*2-elim].

$$\begin{array}{l}(\mathbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet\\ \quad \mathit{value}1 := \mathit{stack}{<}k{>};\\ \quad \mathit{value}2 := \mathit{stack}{<}k+1{>};\\ \quad \mathbf{if}\ \mathit{value}1 \leq \mathit{value}2 \longrightarrow\\ \qquad \ldots\\ \quad [\!]\ \mathit{value}1 > \mathit{value}2 \longrightarrow\\ \qquad \ldots\\ \quad \mathbf{fi})\end{array} \sqsubseteq_A \begin{array}{l}\mathbf{if}\ \mathit{stack}{<}k{>} \leq \mathit{stack}{<}k+1{>} \longrightarrow\\ \qquad \ldots\\ [\!]\ \mathit{stack}{<}k{>} > \mathit{stack}{<}k+1{>} \longrightarrow\\ \qquad \ldots\\ \mathbf{fi}\end{array}$$

Figure 5.43: Rule [cond-*value*1-*value*2-elim]

applied to every method call in the body of *methodName*. These rules eliminate *poppedArgs* and copy the values stored in it into the values passed to the value parameters of the called method. This can be seen in Rule [*poppedArgs*-elim], shown in Figure 5.44, which eliminates *poppedArgs* when associated with method calls arising from a `invokespecial` or `invokestatic` instruction. Rule [*poppedArgs*-sync-elim] and Rule [invokevirtual-*poppedArgs*-elim] are similar, but account for the extra communication before synchronized methods, and the extra *getClassIDOf* communication and multiple targets arising from `invokevirtual` instructions, respectively.

Rule [*poppedArgs*-elim]**.**

$$\begin{array}{l} (\mathbf{var}\, poppedArgs : \mathrm{seq}\, Word \bullet \\ poppedArgs := \langle arg_1, \ldots, arg_n \rangle; \\ M(poppedArgs\, 1, \ldots, poppedArgs\, n)) \end{array} \quad \sqsubseteq_A \quad M(arg_1, \ldots, arg_n)$$

Figure 5.44: Rule [*poppedArgs*-elim]

We also move any actions that are at the start of a loop before the loop condition is checked. Such actions cannot be represented in C without the use of the comma operator, which is not allowed in MISRA-C. The actions are moved, using Law [rec-rolling-rule], so that they are before the start of the loop and at the end of the loop body. This is performed on line 24.

After these rules have been applied to the body of *TPK_handleAsyncEvent*, it has the form shown in Figure 5.45. The effect of these rules can be seen in the fact that values stored in stack slots such as *stack*1 are passed directly to the arguments of called functions. The conditionals also compare stack slots directly and the assignments to those stack slots ($stack1 := var4$ and $stack2 := 10$) have been moved to before the start of the loop and just before the end of the loop, rather than just after the beginning of the loop. The argument to the function is passed in via the *arg*1 variable and assigned to the local variable *var*1. This indirection is unnecessary, and we wish instead to have the argument passed directly into *var*1. We thus perform some final transformations to turn the local variables corresponding to the methods arguments into parameters and eliminate the *arg*1, ..., *arg*<*n*> parameters for the method.

First, we make the first n local variables into parameters using Rule [*var*-parameter-conversion], shown in Figure 5.46. This matches the *Circus* variable blocks representing local variables and stack slots, along with the assignments initialising the first n local variables. The rule moves these assignments and, using the definition of value parameter, converts them into instantiations of value parameters. This is applied to the method's action on line 25.

After local variables have been converted into arguments, we redefine the method action to exclude the parametrised block for the *arg*1, ..., *arg*<*n*> parameters, so that the, now redundant, parameters can be eliminated. This is performed, as with previous redefinitions of method actions, in a separate procedure, RedefineMethodActionToExcludeParameters, defined by Algorithm 20 in Appendix A. This procedure is called on line 26 of Algorithm 12.

After this, the argument parameters *arg*1, ..., *arg*<*n*> are outside the method action and can be eliminated by application of Rule [argument-variable-elimination], shown in Figure 5.47. This rule takes the name of the method action as a parameter and eliminates the argument parameters around the call to the method action, passing their values directly to the method action. It is applied on line 27.

As in the previous section, we only handle methods that do not return a value. Handling methods that do return a value requires additional compilation rules, similar to Rule [*var*-

```
$TPK\_handleAsyncEvent \mathrel{\widehat=} \mathbf{val}\ arg1 : Word \bullet$
  $\mathbf{var}\ var1 var2, var3, var4, var5, var6 : Word \bullet \mathbf{var}\ stack1, stack2, stack3 : Word \bullet$
  $var2 := arg1\ ;\ Poll;$
  $newObject!thread!ConsoleConnectionClassID$
      $\longrightarrow newObjectRet!oid \longrightarrow stack1 := oid\ ;\ Poll;$
  $stack2 := stack1\ ;\ Poll;$
  $stack3 := null\ ;\ Poll;$
  $\cdots$
  $var5 := stack1\ ;\ Poll\ ;\ Poll;$
  $stack1 := var5\ ;\ Poll;$
  $stack2 := 10\ ;\ Poll;$
  $\mu Y \bullet$
    $\mathbf{if}\ stack1 \leq stack2 \longrightarrow Poll;$
        $stack1 := var3\ ;\ Poll;$
        $getClassIDOf!stack1?cid \longrightarrow ConsoleInput\_read(stack1, stack1)\ ;\ Poll;$
        $TPK\_f(stack1, stack1)\ ;\ Poll;$
        $var6 := stack1\ ;\ Poll;$
        $\cdots$
        $stack1 := var5\ ;\ Poll;$
        $stack2 := 10\ ;\ Poll\ ;\ Y$
    $\talloblong\ stack1 > stack2 \longrightarrow \mathbf{Skip}$
    $\mathbf{fi}\ ;\ Poll$
```

Figure 5.45: *TPK_handleAsyncEvent* after its variables have been introduced

Rule [*var*-parameter-conversion]**.**

$$\begin{array}{l}
(\mathbf{var}\ var1, \ldots, var{<}\ell{>} : Word \bullet \\
\mathbf{var}\ stack1, \ldots, stack{<}s{>} : Word \bullet \\
\quad var1 := arg1; \\
\quad \cdots \\
\quad var{<}n{>} := arg{<}n{>}; \\
\quad A)
\end{array}
\sqsubseteq_A
\begin{array}{l}
(\mathbf{val}\ var1, \ldots var{<}n{>} : Word \bullet \\
\mathbf{var}\ var{<}n+1{>}, \ldots, var{<}\ell{>} : Word \bullet \\
\mathbf{var}\ stack1, \ldots, stack{<}s{>} : Word \bullet \\
\quad A)(arg1, \ldots, arg{<}n{>})
\end{array}$$

Figure 5.46: Rule [*var*-parameter-conversion]

Rule [argument-variable-elimination]**.** Given an action name M,

$$\begin{array}{l}
(\mathbf{val}\ arg1, \ldots, arg{<}n{>} : Word \bullet \\
\quad M(arg1, \ldots, arg{<}n{>}))(arg_1, \ldots, arg_n)
\end{array}
\sqsubseteq_A M(arg_1, \ldots, arg_n)$$

Figure 5.47: Rule [argument-variable-elimination]

parameter-conversion] and Rule [argument-variable-elimination], to account for the additional result parameter present in such methods. The result parameter is not replaced with a local variable, since it is only used for returning the value and, as such, does not map onto a specific local variable. Instead, it is simply moved to be grouped with the local variable parameters. The rules on lines 20 to 22 also require separate versions to handle the storing of the returned value in the calling method.

When the variables have been introduced, the model that we obtain has a form that corresponds directly to the C code for each method. This can be seen from Figures 5.48 and 5.49, which show the *Circus* code for *TPK_handleAsyncEvent* and its corresponding C code generated using our prototype implementation of the strategy described in Section 6.3. Note that the *getClassIDOf* communications with the object manager correspond to accesses to C structs representing objects. These structs are introduced in the final stage the strategy, in Section 5.5.

5.4.4 Remove *frameStack* From State

After all methods have been refined to use individual variables, the *frameStack* is no longer used. We can thus eliminate the *frameStack* from the state. This is performed as described in Algorithm 13, which defines the RemoveFrameStackFromState procedure.

Algorithm 13 RemoveFrameStackFromState

```
  apply Law [forwards-data-refinement]([], $FrameStackEliminationCI$)
  apply Law [process-param-elim]($cs$)
```

First, on line 1, we perform a data refinement to remove the *frameStack* from the state. The new state after the data refinement is the empty schema, and the coupling invariant, *FrameStackEliminationCI*, maps all *frameStack* values onto the empty state. Since *frameStack* is no longer used in the process, the only action affected is the state initialisation, which becomes *Skip*. After this, on line 2, we eliminate the *cs* parameter from the process, since it is no longer used. The result is the process $CThr_{bc,cs}(t)$, as shown in Theorem 5.4.1. The only thing remaining to be done is to refine the representation of objects, which is performed in the next stage of the strategy.

5.5 Data Refinement of Objects

The final stage of the compilation strategy introduces the representation of objects in C. Unlike the previous stages of the strategy, this stage operates on the object manager, *ObjMan*, refining it to the $StructMan_{cs}$ process described in Section 4.4.2. Thus, this stage may be summarised by the following theorem.

Theorem 5.5.1 (Data Refinement of Objects).

$$ObjMan(cs) \sqsubseteq StructMan_{cs}$$

The process of refining *ObjMan* is described by Algorithm 14. It begins on line 1 with a data refinement to change the representation of objects used in the interpreter to the C structs used in the final code. The fields in all superclasses of a class are collected together to form the fields used to define its struct. Note that an interface cannot have non-static fields and so, since

```
$TPK\_handleAsyncEvent \mathrel{\widehat=} \mathbf{val}\ var1 : Word \bullet$
    $\mathbf{var}\ var2, var3, var4, var5, var6 : Word \bullet \mathbf{var}\ stack1, stack2, stack3 : Word \bullet Poll;$
    $newObject!thread!ConsoleConnectionClassID$
        $\longrightarrow newObjectRet!oid \longrightarrow stack1 := oid\ ;\ Poll;$
    $stack2 := stack1\ ;\ Poll;$
    $stack3 := null\ ;\ Poll;$
    $ConsoleConnection\_CCinit(stack2, stack3)\ ;\ Poll;$
    $var2 := stack1\ ;\ Poll;$
    $stack1 := var2\ ;\ Poll;$
    $getClassIDOf!stack1?cid$
        $\longrightarrow ConsoleConnection\_openInputStream(stack1, stack1)\ ;\ Poll;$
    $var3 := stack1\ ;\ Poll;$
    $stack1 := var2\ ;\ Poll;$
    $getClassIDOf!stack1?cid$
        $\longrightarrow ConsoleConnection\_openOutputStream(stack1, stack1)\ ;\ Poll;$
    $var4 := stack1\ ;\ Poll;$
    $stack1 := 0\ ;\ Poll;$
    $var5 := stack1\ ;\ Poll\ ;\ Poll;$
    $stack1 := var5\ ;\ Poll;$
    $stack2 := 10\ ;\ Poll;$
    $\mu Y \bullet$
        $\mathbf{if}\ stack1 \leq stack2 \longrightarrow Poll;$
            $stack1 := var3\ ;\ Poll;$
            $getClassIDOf!stack1?cid \longrightarrow$
                $\mathbf{if}\ cid = ConsoleInputClassID \longrightarrow ConsoleInput\_read(stack1, stack1);$
                $\mathbf{fi}\ ;\ Poll;$
            $TPK\_f(stack1, stack1)\ ;\ Poll;$
            $var6 := stack1\ ;\ Poll;$
            $stack1 := var6\ ;\ Poll;$
            $stack2 := 400\ ;\ Poll;$
            $\mathbf{if}\ stack1 \leq stack2 \longrightarrow Poll;$
                $stack1 := var4\ ;\ Poll;$
                $stack2 := var6\ ;\ Poll;$
                $getClassIDOf!stack1?cid \longrightarrow$
                    $\mathbf{if}\ cid = ConsoleOutputClassID \longrightarrow ConsoleOutput\_write(stack1, stack2)$
                    $\mathbf{fi}$
            $[]\ stack1 > stack2 \longrightarrow Poll;$
                $\ldots$
            $\mathbf{fi}\ ;\ Poll;$
            $stack1 := var5\ ;\ Poll;$
            $stack2 := 1\ ;\ Poll;$
            $stack1 := stack1 + stack2\ ;\ Poll;$
            $var4 := stack1\ ;\ Poll$
            $stack1 := var5\ ;\ Poll;$
            $stack2 := 10\ ;\ Poll\ ;\ Y$
        $[]\ stack1 > stack2 \longrightarrow \mathbf{Skip}$
        $\mathbf{fi}\ ;\ Poll$
```

Figure 5.48: *TPK_handleAsyncEvent* at the end of the Introduce Variables step

```
void TPK_handleAsyncEvent(int32_t var1) {
  int32_t var2, var3, var4, var5, var6;
  int32_t stack1, stack2, stack3;
  stack1 = newObject(ConsoleConnectionID);
  stack2 = stack1;
  stack3 = 0;
  ConsoleConnection_init(stack2, stack3);
  var2 = stack1;
  stack1 = var2;
  if (((Object*) ((uintptr_t)stack1))->classID == ConsoleConnectionID) {
    ConsoleConnection_openInputStream(stack1, & stack1);
  }
  var3 = stack1;
  stack1 = var2;
  if (((Object*) ((uintptr_t)stack1))->classID == ConsoleConnectionID) {
    ConsoleConnection_openOutputStream(stack1, & stack1);
  }
  var4 = stack1;
  stack1 = 0;
  var5 = stack1;
  stack1 = var5;
  stack2 = 10;
  while (stack1 <= stack2) {
    stack1 = var3;
    if (((Object*) ((uintptr_t)stack1))->classID == ConsoleInputID) {
      ConsoleInput_read(stack1, & stack1);
    }
    TPK_(stack1, & stack1);
    var6 = stack1;
    stack1 = var6;
    stack2 = 400;
    if (stack1 <= stack2) {
      stack1 = var4;
      stack2 = var6;
      if (((Object*) ((uintptr_t)stack1))->classID == ConsoleOutputID) {
        ConsoleOutput_write(stack1, stack2);
      }
    } else {
      ...
    }
    stack1 = var5;
    stack2 = 1;
    stack1 = stack2 + stack1;
    var5 = stack1;
    stack1 = var5;
    stack2 = 10;
  }
}
```

Figure 5.49: The C code corresponding to *TPK_handleAsyncEvent*

we require that the inputs to the compilation strategy pass the standard checks performed on Java class files, a class's objects only inherit fields from its true superclasses, even though our superclass relation includes implemented interfaces.

As an example, we present the struct for objects of the `TPK` class, *TPKObj*, below. Since `TPK` declares no fields of its own (as indicated by the empty *fields* set for *TPK* in Figure 5.5 from Section 5.3.1), it only includes fields from the superclasses of `TPK` (shown in Figure 5.6). Its fields are thus those of `ManagedEventHandler`, since `AperiodicEventHandler` does not add any information to the base information for an event handler. These fields are in addition to the *classID* field, which is contained in every object's struct and identifies the class of the object. The other fields are the *threadID*, *backingStoreSize*, *allocAreaSize* and *stackSize* fields from the *ManagedEventHandler* class information structure. These provide space in which the information about the event handler may be stored by the *Launcher* during mission startup.

```
─ TPKObj ─────────────────────────────
│ classID : ClassID
│ threadID : Word
│ backingStoreSize : Word
│ allocAreaSize : Word
│ stackSize : Word
└─────────────────────────────────────
```

Similar types are created for *ManagedEventHandler* and *AperiodicEventHandler*, with the same fields, and named *ManagedEventHandlerObj* and *AperiodicEventHandlerObj* respectively. This means that *TPKObj* values can be converted to those types, since they contain fields of the same name and type. Other aperiodic event handlers may have additional fields to store data specific to them. Their object types can be converted to *AperiodicEventHandlerObj* by simply discarding the additional fields. The struct types for each class are collected together into an *ObjectStruct* type, shown below.

$$
\begin{aligned}
&\mathit{ObjectStruct} ::= \\
&\quad \mathit{TPKCon}\langle\!\langle \mathit{TPKObj} \rangle\!\rangle \mid \\
&\quad \mathit{AperiodicEventHandlerCon}\langle\!\langle \mathit{AperiodicEventHandlerObj} \rangle\!\rangle \mid \\
&\quad \mathit{ManagedEventHandlerCon}\langle\!\langle \mathit{ManagedEventHandlerObj} \rangle\!\rangle \mid \cdots
\end{aligned}
$$

The values of *ObjectStruct* that can be converted to *ManagedEventHandlerObj* can be cast to it by the function *castManagedEventHandler*. We also define functions for performing a combined cast and field update and collect the static fields from all classes into a *StaticFields* structure, as described in Section 4.4.2.

Algorithm 14 Data Refinement of Objects

```
1: apply Law [forwards-data-refinement]($StructManState$, $ObjectCI$)
2: apply Rule [refine-$NewObject$] to $NewObject$
3: apply Rule [refine-$GetField$] to $GetField$
4: apply Rule [refine-$PutField$] to $PutField$
5: apply Rule [refine-$GetStatic$] to $GetStatic$
6: apply Rule [refine-$PutStatic$] to $PutStatic$
7: apply Law [process-param-elim]($cs$)
```

These types and functions are all used in the struct manager, so we introduce them before applying the data refinement. Note that, since the types are Z data types, they do not need to be declared in a process and so we do not need to introduce them by refinement of *ObjMan*.

The state resulting from the data refinement on line 1 is *StructManState*, shown below, which uses the object struct types described above.

$$
\begin{array}{|l}
\textit{StructManState} \\
\hline
\textit{BackingStoreManager} \\
\textit{objects} : \textit{ObjectID} \pfun \textit{ObjectStruct} \\
\textit{staticClassFields} : \textit{StaticFields} \\
\hline
\textit{backingStoreMap} \text{ partition } \operatorname{dom} \textit{objects} \\
\hline
\end{array}
$$

As mentioned in Section 4.4.2, *StructManState* is very similar to *ObjManState*. It contains *BackingStoreManager*, which is the same as in *ObjManState* since the management of backing stores is unaffected by the compilation strategy. The *objects* map is similar to that in *ObjManState*, but it maps to the *ObjectStruct* type, rather than the *Object* type. The component *staticClassFields*, in *StructManState*, is of the *StaticFields* type, rather than the map from fields to their values in *ObjManState*, since we have a known set of static fields, having been supplied with the *cs* map.

The data refinement is described by the coupling invariant *ObjectCI*, shown in Figure 5.50, which relates *ObjManState* to *StructManState*. It is presented as a template to be instantiated by the identifiers of the classes in *cs* and their corresponding fields, in much the same format as for the schemas in Section 4.4.2. This equates the *BackingStoreManager* fields in *ObjManState* with those in *StructManState*, since management of backing stores is unaffected by the data refinement.

The functions *objects* for both the abstract and concrete states have the same domain, since the set of objects does not change, merely their representation. The representation of each object in *objects* after the data refinement is determined by the class identifier in its *class* information before the refinement. The object is of the struct type for the class identifier. For example, *TPKClassID* is the class identifier corresponding to the class information of `TPK`, since that is the identifier in the *constantPool* of the *TPK* structure (Figure 5.5) that corresponds to its *this* value. Thus, any object of that type will have the corresponding class information stored and so is refined to an *ObjectStruct* value using the *TPKCon* constructor and containing a value of the *TPKObj* type shown above.

The fields of the structure are taken from the values corresponding to each field identifier in the object's information before the data refinement. These fields are guaranteed to correspond to the fields listed in the object's class information (including the fields from its superclasses) by the invariant of *ObjManState*. Similarly, the fields in *staticClassFields* map directly onto fields in the *StaticFields* structure, with the set of fields guaranteed to be the same by the invariant of *StaticFieldsInfo*.

ObjectCI is based on equating the information in the old model with the fields of the object structs in the new model, and so it describes a functional data refinement from which we can calculate the form of the actions in the resultant model. However, the direct application of this refinement yields actions in a form that does not directly correspond to the representation of the semantics of C structs that we desire. We thus apply some additional compilation rules on lines 2 to 6, to refine the actions of the process to the correct form.

$$
\begin{array}{l}
\textit{ObjectCI} \\
\hline
\quad \textit{ObjManState} \\
\quad \textit{StructManState}_1 \\
\hline
\quad \textit{backingStoreMap} = \textit{backingStoreMap}_1 \\
\quad \textit{backingStoreStacks} = \textit{backingStoreStacks}_1 \\
\quad \textit{rootBS} = \textit{rootBS}_1 \\
\quad \mathrm{dom}\ \textit{objects} = \mathrm{dom}\ \textit{objects}_1 \\
\quad \forall x : \mathrm{dom}\ \textit{objects} \bullet \\
\qquad (\textit{thisClassID}\ ((\textit{objects}\ x).\textit{class}) = {<}\textit{classID}_1{>} \Rightarrow \\
\qquad\quad \textit{objects}_1\ x = {<}\textit{classID}_1{>}\textit{Con}\ \lblot \\
\qquad\qquad \textit{classID} == {<}\textit{classID}_1{>}, \\
\qquad\qquad {<}\textit{fieldID}_{1,1}{>} == (\textit{objects}\ x).\textit{fields}\ {<}\textit{fieldID}_{1,1}{>}, \\
\qquad\qquad\quad \ldots \\
\qquad\qquad {<}\textit{fieldID}_{1,m_1}{>} == (\textit{objects}\ x).\textit{fields}\ {<}\textit{fieldID}_{1,m_1}{>} \\
\qquad\quad \rblot) \land \\
\qquad\quad \ldots \\
\qquad (\textit{thisClassID}\ ((\textit{objects}\ x).\textit{class}) = {<}\textit{classID}_n{>} \Rightarrow \\
\qquad\quad \textit{objects}_1\ x = {<}\textit{classID}_n{>}\textit{Con}\ \lblot \\
\qquad\qquad \textit{classID} == {<}\textit{classID}_n{>}, \\
\qquad\qquad {<}\textit{fieldID}_{n,1}{>} == (\textit{objects}\ x).\textit{fields}\ {<}\textit{fieldID}_{n,1}{>}, \\
\qquad\qquad\quad \ldots \\
\qquad\qquad {<}\textit{fieldID}_{n,m_n}{>} == (\textit{objects}\ x).\textit{fields}\ {<}\textit{fieldID}_{n,m_n}{>} \\
\qquad\quad \rblot) \\
\quad \textit{staticClassFields}_1.{<}\textit{classID}_1{>}\_{<}\textit{staticfieldID}_{1,1}{>} = \\
\qquad \textit{staticClassFields}\ ({<}\textit{classID}_1{>}, {<}\textit{staticfieldID}_{1,1}{>}) \\
\qquad \ldots \\
\quad \textit{staticClassFields}_1.{<}\textit{classID}_1{>}\_{<}\textit{staticfieldID}_{1,\ell_1}{>} = \\
\qquad \textit{staticClassFields}\ ({<}\textit{classID}_1{>}, {<}\textit{staticfieldID}_{1,\ell_1}{>}) \\
\qquad \ldots \\
\quad \textit{staticClassFields}_1.{<}\textit{classID}_n{>}\_{<}\textit{staticfieldID}_{n,1}{>} = \\
\qquad \textit{staticClassFields}\ ({<}\textit{classID}_n{>}, {<}\textit{staticfieldID}_{n,1}{>}) \\
\qquad \ldots \\
\quad \textit{staticClassFields}_1.{<}\textit{classID}_n{>}\_{<}\textit{staticfieldID}_{n,\ell_n}{>} = \\
\qquad \textit{staticClassFields}\ ({<}\textit{classID}_n{>}, {<}\textit{staticfieldID}_{n,\ell_n}{>}) \\
\hline
\end{array}
$$

Figure 5.50: The *ObjectCI* schema, which is the coupling invariant between *ObjManState* and *StructManState*

Rule [refine-*NewObject*].

$$
\begin{array}{l}
\mathbf{var}\ \mathit{thread} : \mathit{ThreadID};\ \mathit{classID} : \mathit{ClassID} \bullet \\
\mathbf{var}\ \mathit{objectID} : \mathit{ObjectID};\ \mathit{class} : \mathit{Class} \bullet \\
\mathit{newObject}?t?c \longrightarrow \mathit{thread}, \mathit{classID} := t, c; \\
\big(\mathit{GetObjectClassInfo}\big); \\
\mathit{AllocateObject}(\\
\qquad \mathit{thread}, \mathit{sizeOfObject}\ \mathit{class}, \mathit{objectID}); \\
\big(\mathit{StructManObjectInit}\big); \\
\mathit{newObjectRet}!\mathit{objectID} \longrightarrow \mathbf{Skip}
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
\mathbf{var}\ \mathit{objectID} : \mathit{ObjectID} \bullet \\
\mathit{newObject}?\mathit{thread}?\mathit{classID} \longrightarrow \\
\mathbf{if}\ \mathit{classID} = \mathord{<}\mathit{classID}_1\mathord{>} \longrightarrow \\
\quad \mathit{AllocateObject}(\\
\qquad \mathit{thread}, \\
\qquad \mathit{sizeof}\mathord{<}\mathit{classID}_1\mathord{>}\mathit{Obj}, \\
\qquad \mathit{objectID}); \\
\quad \big(\mathit{StructMan}\mathord{<}\mathit{classID}_1\mathord{>}\mathit{ObjInit}\big); \\
\quad \vdots \\
\ [\!]\ \mathit{classID} = \mathord{<}\mathit{classID}_n\mathord{>} \longrightarrow \\
\quad \mathit{AllocateObject}(\\
\qquad \mathit{thread}, \\
\qquad \mathit{sizeof}\mathord{<}\mathit{classID}_n\mathord{>}\mathit{Obj}, \\
\qquad \mathit{objectID}); \\
\quad \big(\mathit{StructMan}\mathord{<}\mathit{classID}_n\mathord{>}\mathit{ObjInit}\big); \\
\mathbf{fi}\ ;\ \mathit{newObjectRet}!\mathit{objectID} \longrightarrow \mathbf{Skip}
\end{array}
$$

where, for all $k \in 1 \mathinner{..} n$,

$$
\begin{array}{l}
\exists\, \Delta \mathit{Class} \mid \Xi \mathit{Class} \setminus (\mathit{fields}, \mathit{fields}') \bullet \\
\quad \theta\ \mathit{Class} = \mathit{cs}\,\mathord{<}\mathit{classID}_k\mathord{>} \land \\
\quad \mathit{fields}' = \bigcup\{\mathit{cid} : \operatorname{dom} \mathit{cs} \mid (\mathord{<}\mathit{classID}_k\mathord{>}, \mathit{cid}) \in \mathit{subclassRel}\ \mathit{cs} \bullet (\mathit{cs}\ \mathit{cid}).\mathit{fields}\} \land \\
\quad \mathit{sizeof}\mathord{<}\mathit{classID}_k\mathord{>}\mathit{Obj} = \mathit{sizeOfObject}\,(\theta\ \mathit{Class}\,')
\end{array}
$$

Figure 5.51: Rule [refine-*NewObject*]

An example of such a rule is Rule [refine-*NewObject*], shown in Figure 5.51. It operates on the body of the *NewObject* action, which has the form on the left-hand side of the rule after the application of the data refinement. In this rule, we represent by the schema *StructManObjectInit* the result of applying the data refinement to the *ObjManObjectInit* schema. The rule splits this data operation into separate operations defined by simpler schemas, which can be found in Appendix B of the extended version of this thesis [13], initialising each of the object struct types, offering a choice over each class identifier in *cs* to determine which one should be used. The *AllocateObject* action that communicates with the memory manager to allocate space for the object is also supplied with constants indicating the size of each object struct type, rather than determining it from the class information in *cs*. This means the *GetObjectClassInfo* schema, which determines the class information, can also be removed, eliminating reliance on *cs* in the action.

Rule [refine-*GetField*], Rule [refine-*PutField*], Rule [refine-*GetStatic*] and Rule [refine-*PutStatic*], which refine the *GetField*, *PutField*, *GetStatic* and *PutStatic* actions respectively, are similar to Rule [refine-*NewObject*], refining the data operations that result from the data refinement to choices over class and field identifiers received on the channels for the operations. These rules can be found in Appendix A. Note that, while the *Init* action of *ObjMan* is refined in this stage, it does not require a separate rule, since the necessary refinement is performed by the data refinement alone.

After the refinement has been performed, we can eliminate the cs parameter of the process via an application of Law [process-param-elim] on line 7. This completes the transformation of $ObjMan(cs)$ into $StructMan_{cs}$. After this, the model corresponds completely to the C code.

5.6 Proof of Main Theorem

The three stages of our strategy, taken together, refine the abstract interpreting CEE described in Section 4.3 to the concrete C CEE described in Section 4.4. This can be seen from the proof of Theorem 5.1.1, shown below.

Proof [Theorem 5.1.1]**.**

$$
\begin{array}{ll}
\quad CEE(bc, cs, instCS, sid, initOrder) & \\
= & \text{[Definition of } CEE\text{]} \\
\quad ObjMan(cs) \parallel Interpreter(cs, bc, instCS) \parallel Launcher(sid, initOrder) & \\
= & \text{[Definition of } Interpreter\text{]} \\
\quad ObjMan(cs) \parallel & \\
\qquad (\parallel t : ThreadID \setminus \{idle\} \llbracket\, ThrChans(t) \rrbracket \bullet Thr(bc, cs, instCS, t)) \parallel & \\
\qquad Launcher(sid, initOrder) & \\
\sqsubseteq & \text{[Theorem 5.3.1]} \\
\quad ObjMan(cs) \parallel & \\
\qquad (\parallel t : ThreadID \setminus \{idle\} \llbracket\, ThrChans(t) \rrbracket \bullet ThrCF_{bc,cs}(cs, t)) \parallel & \\
\qquad Launcher(sid, initOrder) & \\
\sqsubseteq & \text{[Theorem 5.4.1]} \\
\quad ObjMan(cs) \parallel & \\
\qquad (\parallel t : ThreadID \setminus \{idle\} \llbracket\, ThrChans(t) \rrbracket \bullet CThr_{bc,cs}(t)) \parallel & \\
\qquad Launcher(sid, initOrder) & \\
= & \text{[Definition of } CProg_{bc,cs}\text{]} \\
\quad ObjMan(cs) \parallel CProg_{bc,cs} \parallel Launcher(sid, initOrder) & \\
\sqsubseteq & \text{[Theorem 5.5.1]} \\
\quad StructMan_{cs} \parallel CProg_{bc,cs} \parallel Launcher(sid, initOrder) &
\end{array}
$$

□

The correctness of this proof rests on the correctness of theorems for each stage of the strategy. The compilation strategy forms the proofs of these theorems and it is composed of applying compilation rules. The correctness of the compilation rules is, in turn, ensured by their proofs in terms of algebraic laws that are known to be correct.

5.7 Final Considerations

In this chapter, we have presented our compilation strategy from an interpreting SCJVM to our model of C code. While our compilation strategy proves the correctness of the compilation, there are further optimisations that may be performed on the output of the strategy.

One example of such an optimisation is the removal of the unnecessary choice offered in virtual method calls with only one possible target. Such choices are made using the class identifier of the object, which, in our model, is obtained via communication with the struct manager on the *getClassIDOf* channel. The removal of the choice requires the removal of this communication, which is a refinement all the processes that participate in this communication. It requires collapsing the parallelism between these processes and using the fact that the communication is hidden to remove it. This is not performed in our strategy, since each stage of the strategy operates only on a single process, but is a relatively straightforward optimisation that could be added as an extension of the strategy in future work.

A further consideration is that Z schema bindings represent an unordered collection of fields, whereas C structs define the order in which their fields are stored in memory. This means that, while our struct manager model defines what fields must be in each object's struct type, it does not specify the order of those fields and so is still somewhat more abstract than the C code itself. This can be addressed by a further data refinement to a representation using Z sequences. Field names for each struct type would then be associated with offsets into these sequences, as field names in C are associated with offsets in the struct's memory. We have not performed such a data refinement as part of the strategy, since we believe the form of the struct manager is sufficiently clear to implementers, although there has to be a choice of ordering for the C structs when implementing.

We expect other optimisations to be performed by the C compiler that compiles the output of the strategy. The correctness of such optimisations is part of verification of the C compiler and thus outside the scope of our work. However, some optimisations could be integrated into the strategy as part of future work. An example is the elimination of unnecessary assignments, such as on line 19 of the code in Figure 5.49. There, `stack1` is used as an intermediate variable to set `var5` and is not otherwise used before it is overwritten on line 21. These assignments are removed by optimising C compilers, and so are not removed by our strategy, or the icecap HVM, but could be removed in order to produce clearer C code.

Other possible directions for future work extending the strategy include weakening the assumptions described in Section 5.2. Our definition of a structured program is slightly stronger than the structural requirements imposed by MISRA-C, which permits a single exit from the middle of a loop in addition to the condition at the start or end of the loop. This means loops may have two exits in MISRA-C, whereas our strategy only accounts for loops with a single exit point. The strategy could be modified to allow for loops with two exit points by adding new rules, similar to Rule [`while`-loop-intro1], to introduce such loops, having two conditionals to allow for exit from the loop.

We do not model and handle integer overflow in the strategy due to the fact that it is not handled in icecap, instead requiring the SCJ programmer to ensure that their code does not include overflows. A possible extension of the strategy would be to model the overflow behaviour of the JVM in the bytecode interpreter and refine it to C code that enforces that overflow behaviour by checking if overflow would occur before performing an operation. This would create extra checks in many places where they would not be necessary, but such checks could be removed in places where overflow can be proved not to occur.

While we do not allow recursion due to its potential for stack overflow, there may be a few cases in which recursion can be shown to be bounded and hence safe. The strategy could be extended to handle such cases, requiring rules for introducing loops caused by method calls, and separating the resulting recursions in recursive actions to represent recursive methods.

Recursion with more than one method involves a similar approach, introducing nested loops and creating mutually recursive actions.

In summary, there are many optimisations and extensions that are possible. What we have achieved, however, is a formal account of a compilation strategy that addresses all the central concerns involved in transforming SCJ bytecode to a higher-level language like C. Correctness of the compiled code is established by construction. In the next chapter, we discuss in more detail how the correctness of the strategy is assured and evaluate it through consideration of some examples.

Chapter 6

Evaluation

In this chapter, we evaluate our model and compilation strategy, using several approaches. The aim of our work is the construction of a correct compilation strategy and so our evaluation in this chapter focusses on establishing the correctness of the transformations performed in the compilation strategy. First, we consider what assurances can be gained from mechanisation of the model and proofs of the compilation rules. In addition, we compare code produced by our strategy to that produced by icecap, using some examples to evaluate the strategy. We note, finally, that the process of constructing the model already embeds important validation effort, via numerous reviews of the standard, and close interaction with the standardisation committee, which led to some changes to the standard.

For clarity, we recall what we have presented thus far. We have developed a model of an interpreting SCJVM, covering both the SCJVM services and the core execution environment. This model has been written using Community Z Tools (CZT), so that it is machine readable, although the nature of the checks that can be performed on it are limited by the capabilities of CZT. We discuss this in more detail in Section 6.1. We have also developed a compilation strategy for translating SCJ bytecode in the interpreter model to a representation of C code. Since there is not yet a sufficiently powerful automated proof assistant for *Circus*, the rules and their corresponding proofs are hand-written, although some of the laws used in the proofs have been proved using an automated proof assistant. The proofs of the compilation rules and the sources of the laws used in them are discussed in Section 6.2. We also add that we have developed a prototype implementation of the compilation strategy that takes in SCJ class files and outputs both C code and the *Circus* models resulting from the compilation strategy. This prototype implementation is discussed in Section 6.3 and then, in Section 6.4 we evaluate the strategy by applying the prototype to some examples. Finally, we conclude in Section 6.5.

6.1 Mechanisation of Models

The correctness of our compilation strategy relies on the correctness of the models used as input to the compilation strategy. Their correctness relies on the inputs to the models meeting the assumptions made in Section 5.2. If these assumptions are not met, then the behaviour of model is not correct and the compilation strategy cannot be applied. For example, if the sequence of instructions in the program causes the operand stack to overflow the maximum stack size, the invariant of *StackFrame* is violated and the program's behaviour is chaotic. Our compilation strategy cannot be applied to such a program, since no stack slots are created beyond the maximum stack size to handle such a situation in the strategy, and it is not clear what the expected C code would be.

As discussed in Section 4.5, the fact that the models are written in CZT ensures they have correct syntax and types. CZT performs this checking continuously and flags up errors as they occur, so they can be quickly corrected during the writing of the models.

We have also performed some proofs on the Z schemas defining the semantics of the bytecode instructions, using Z/EVES 2.4.1 with CZT as its user interface. There are two main groups of results. The first is domain check proofs, ensuring partial functions are not applied outside their domain. These are proof obligations generated by Z/EVES, and so do not have corresponding theorems stated. These proofs are not required for schemas that do not directly reference partial functions.

The second group of results is precondition proofs. These require that a final state exists for the schema, which ensures that the requirements of the schema are not contradictory. Stating and proving these theorems also extracts the preconditions of the operations, since those must be stated as assumptions of the theorems.

The preconditions we have found include those required to avoid operand stack overflows and underflows, that local variable indices are within the range of the local variable array, and that program-address updates do not go outside of the current method's bytecode array. These conditions are ensured by standard JVM bytecode verification, which we assume inputs to the strategy pass. The existence of at least one stack frame is also required for bytecode instructions to execute, and this property is ensured by the condition on the loop in the *Running* action.

A further precondition required by the interpreter operations is that the value *cs* is such that the class and method in which a program address occurs is unique. This condition is required to ensure that the current class and method can be uniquely determined from the value of *pc*. This is required by the invariant of *InterpreterState*, but need only be fulfilled as a precondition when a new stack frame is created, since it can be ensured from the invariant on the initial state for the other operations. This condition on *cs* is reasonable since the bytecode instructions for each method should be at separate addresses in *bc*.

The statements of the theorems proved can be found, with their corresponding proofs, in Appendix F of the extended version of this thesis [13]. We have also proved various additional lemmas in the course of constructing these proofs. Some of these are general facts that could be of use in other theorems. They are listed along with the theorems, in Appendix E of the extended version of this thesis [13].

6.2 Proofs of Laws

The correctness of our compilation strategy is ensured by the correctness of the individual compilation rules. We prove these rules in terms of algebraic laws, whose correctness is known. This gives assurance that no step of the compilation strategy involves applying a transformation that changes the semantics of the input program.

We adopt an algebraic style of proof, in which the algebraic laws are applied one-by-one to transform the left-hand-side of a rule into its right-hand-side. This ensures that the term obtained in each step of the proof is shown to be a refinement of, or equal to, that of the previous step, by application of a known law. The overall proof then follows from the transitivity of refinement. Thus, every step of the proof is justified formally and this can be easily seen from the layout of the proof.

Overall, there are 91 compilation rules in our strategy, all of which are presented in Appendix A. Of these, we have completed hand-written proofs for 46 rules. In particular, we have proved all the rules used in Algorithms 2, 3, 4, 7, 11 and 16. In the other algorithms, we have proved at least one rule of each type. Proofs of other rules of the same type are similar. Note that Algorithms 1, 5, 8, 9, 13, 15, 17 and 20 only consist of applications of algebraic laws, which we discuss below, and references to other algorithms. In Algorithm 6, we have proved Rule [refine-invokestatic], Rule [refine-invokevirtual] and Rule [resolve-normal-method]. In Algorithm 10 we have proved Rule [refine-*HandleAreturnEPC*-empty-*frameStack*]. In Algorithm 12, we have proved Rule [refine-*PutfieldSF*] and Rule [*HandleAloadSF*-simulation], and, finally, in Algorithm 14, we have proved Rule [refine-*NewObject*]. The rules used in Algorithms 18 and 19 are assumption distribution rules similar to those used in Algorithm 16. Note that the rules we have not written proofs for are similar to those already proved, and mainly concern movement of data. The more challenging rules that transform control flow have been proved, particularly the loop and conditional introduction rules, which are not applied by icecap and so cannot be checked by comparison to icecap's output.

There are a total of 80 laws used in the proofs of the compilation rules. These laws come from various sources. There are 35 laws that are taken from [88], and 8 laws taken from [78]. Those laws have already been proved as part of those works, and so can be safely reused. We have also used 3 ZRC laws from [24], which can be applied to *Circus* since the semantics of ZRC are compatible with those of *Circus*, by Theorem 4.3 from [88]. Standard least-fixed-point laws, stated in [47] are also applied to *Circus* recursion, since it defined using least-fixed-points, and this yields a further 6 laws. Some of the laws follow as a trivial consequence of the definitions given in these sources, such as Law [action-intro], which follows from the definition of process refinement, which does not reference actions not used in the main action of a process. A further 8 laws are obtained from simple combinations of the other laws.

We have proved 20 laws using the proof assistant Isabelle [86] with its implementation of UTP [37]. The constructs supported by that implementation limit the types of laws that may be proved, but we have proved several laws relating to conditionals, assumptions, and assignment. In the case of conditionals, we contributed an implementation of *Circus* conditionals to Isabelle/UTP. This has allowed us to prove laws more general than those that have been proved previously, since previous laws have used the fact that conditionals can be converted to external choice, which requires that the guards be disjoint and provide complete coverage. We require these more general laws to perform transformation of the *Running* action during the elimination of program counter, since not all program counter values have a corresponding bytecode instruction, so we cannot ensure coverage. Our work on this has now been integrated into Isabelle/UTP itself.

The proofs of the compilation rules occupy a total of approximately 300 pages. The number of laws required to prove each compilation rule varies between the compilation rules. As an example, Rule [refine-invokestatic], consists of a total of 31 applications of 16 distinct laws. This may be regarded as a typical proof, but some rules, particularly the assumption distribution rules, follow from specialisations of a single law, while others, such as Rule [*HandleInstruction*-refinement], are large proofs involving multiple cases. Some of the proofs make use of auxiliary lemmas that allow part of the proof to be shared between proofs. This is particularly the case for elimination of program counter rules, where we must unroll the *Running* loop as part of their proofs. This reuse of lemmas makes it challenging to count the total number of laws used in these proofs, so we do not provide detailed information on lengths of proofs here.

There are 9 algebraic laws that are applied directly in our strategy, in addition to the 91 compilation rules. These may be found at the end of Appendix A, after the compilation rules specific to each stage of the strategy. A full list of the algebraic laws used in this thesis, including both those used in our compilation strategy and those used in the proofs of the compilation rules, can be found in Appendix E of the extended version of this thesis [13].

6.3 Prototype Implementation of the Compilation Strategy

In addition to proving the individual compilation rules, it also is useful to be able to automatically generate the code resulting from the strategy in order to validate it. This allows for consideration of the issues involved in handling actual SCJ programs and shows how the strategy as a whole fits together to produce the final code. It also facilitates the consideration of examples, which provide additional validation of the strategy.

We have thus created a simple prototype to transform SCJ class files to the corresponding *Circus* models generated by the strategy, and their corresponding C code. This prototype is written in Java, using the Apache bytecode emulation library for reading class files so that real output from the standard Java compiler can be used directly. It outputs the *Circus* code for the *CThr* process that results from applying the compilation strategy to the input files. We focus on this part of the C code model, and the first two stages of the strategy that generate it, as it is quite complex and so most benefits from review of the code produced. The C code that corresponds to the *CThr* process is also generated by our prototype, by traversing the *Circus* abstract syntax tree to output the C code corresponding to each *Circus* construct.

The data refinement of memory is comparatively simple, since it just involves collecting the fields for each class and producing the corresponding *Circus* code from the strategy. Its correctness is sufficiently ensured by the correctness of the compilation rules, so we do not handle it in our prototype.

To ensure we get the most benefit from our prototype, we follow the strategy and the form of the compilation rules as closely as possible in its design, shown in Figure 6.1. Our implementation of the compilation strategy validates our reasoning in designing it, since the code generated for the examples has the expected form matching that of the icecap compiler.

Some of the classes used in our implementation and the relationships between them are shown in Figure 6.1. Our prototype begins by reading each input class file and extracting the information into `ClassModel` and `BytecodeModel` classes. `ClassModel` represents the *Class* type from our model and makes available all the information represented in that type. `BytecodeModel` is an abstract class whose subclasses represent individual bytecode instructions; it represents the *Bytecode* type from our model. The set of `ClassModel` structures and array of `BytecodeModels` are collected together into a `Model`, representing the inputs to the compilation strategy.

The application of the first stage of the compilation strategy to a `Model` is initiated by invocation of its `doEliminationOfProgramCounter()` method. This returns a `ThrCFModel` object, which represents the *ThrCF* process generated from the inputs represented by the `Model`. The `doEliminationOfProgramCounter()` method applies each step of Algorithm 1. It begins by replacing each bytecode instruction with the *Circus* actions that result from applying bytecode expansion to it, as described in Algorithm 2. We represent *Circus* actions by subtypes of an abstract class `CircusAction`. These subtypes represent both general *Circus* constructs such as

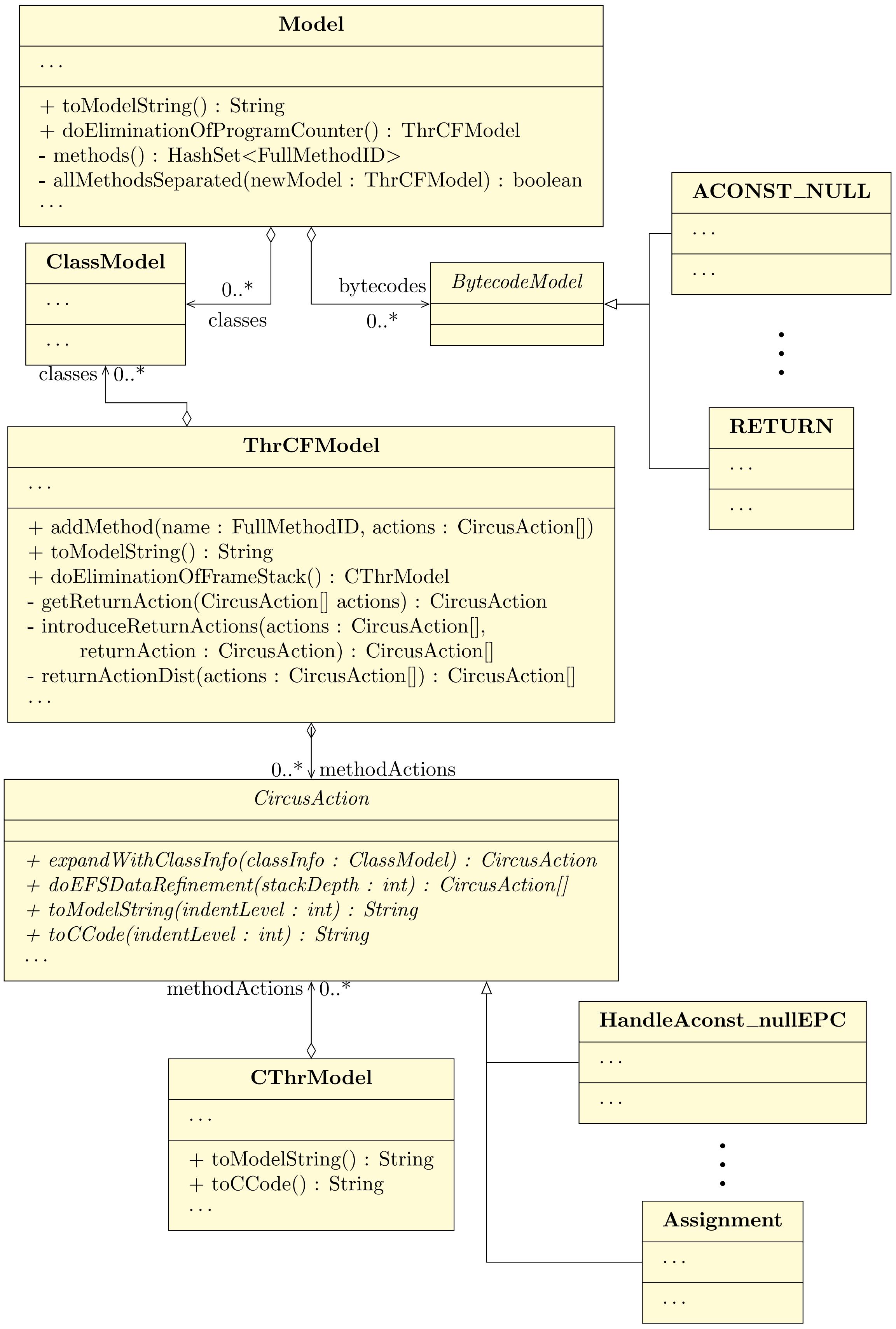

Model
...
+ toModelString() : String
+ doEliminationOfProgramCounter() : ThrCFModel
- methods() : HashSet<FullMethodID>
- allMethodsSeparated(newModel : ThrCFModel) : boolean
...
ClassModel
...
...
0..*
classes
bytecodes
0..*
BytecodeModel
ACONST_NULL
...
...
RETURN
...
...
classes
0..*
ThrCFModel
...
+ addMethod(name : FullMethodID, actions : CircusAction[])
+ toModelString() : String
+ doEliminationOfFrameStack() : CThrModel
- getReturnAction(CircusAction[] actions) : CircusAction
- introduceReturnActions(actions : CircusAction[], returnAction : CircusAction) : CircusAction[]
- returnActionDist(actions : CircusAction[]) : CircusAction[]
...
0..*
methodActions
CircusAction
+ expandWithClassInfo(classInfo : ClassModel) : CircusAction
+ doEFSDataRefinement(stackDepth : int) : CircusAction[]
+ toModelString(indentLevel : int) : String
+ toCCode(indentLevel : int) : String
...
methodActions
0..*
CThrModel
...
+ toModelString() : String
+ toCCode() : String
...
HandleAconst_nullEPC
...
...
Assignment
...
...

Figure 6.1: Class diagram for our implementation of the compilation strategy

variable blocks, conditionals and assignment, and references to specific actions in our model, such as the *Handle∗EPC* actions.

The sequences of actions produced by bytecode expansion are placed into an array of arrays of `CircusAction`s, representing the branches of the choice over *pc* in *Running*. We test the types of the actions in these sequences to check if they match the compilation rules of the strategy, and update the sequence of actions in a branch accordingly, in order to perform the introduction of sequential composition (Algorithm 3), introduction of loops and conditionals (Algorithm 4), and method resolution (Algorithm 6).

We also construct a control-flow graph, which we use to guard the application of some rules as indicated in the strategy, and which is reconstructed after the application of a compilation rule. The sequence of actions corresponding to the entry point of a method whose control-flow graph consists of a single node are added to the `ThrCFModel` during method separation (Algorithm 5), with their *pc* assignments removed when they are added, to produce the result of Algorithm 8. The refine main actions step (Algorithm 7) operates on the *Started* and *MainThread* actions, which have a known form, so we simply output the form resulting from this step, instantiated with the method names collected in the strategy.

The application of the elimination of frame stack stage to the `ThrCFModel` is performed by its `doEliminationOfFrameStack()` method, which returns a `CThrModel` representing the *CThr* process generated after this stage. In our implementation we apply the rules of this stage by traversing the actions of each method, checking for actions that match the form of the rules. Rules that operate on sequences of more than one action are applied by private methods of `ThrCFModel`, whereas those that affect only a single action are applied by methods of the `CircusAction` classes. We group together the application of similar rules in some of these methods.

The removal of launcher returns (Algorithm 10) is performed by first obtaining the return action with a `getReturnAction()` method, which corresponds to the ReturnAction function referenced on line 3 of Algorithm 10. The return action is then introduced after infinite loops by a method `introduceReturnActions()`, which performs the exhaustive application of Law [rec-action-intro] on line 4, and distributed using a method `returnActionDist()`, which performs the exhaustive application of Rule [conditional-dist] on line 5. The remainder of this step is upon the *ExecuteMethod*, *Started* and *MainThread* actions, whose forms are known, so we simply output the resultant forms for them at the end of the application of the strategy. The return actions within the body of a method are refined to the corresponding data operations by this step so we take them to refer to those data operations in subsequent steps. Although the return actions are distributed outside the method actions in this step, they are moved back inside the method actions in the next step, so we do not perform this moving in the implementation.

During the localise stack frames step (Algorithm 11), the data refinement on line 1 of Algorithm 11 only affects the definition of the process' data operations, not the *Circus* code for the methods of our program, so it does not need to be explicitly performed in our implementation. We instead begin localising the stack frames by calculating the number of arguments for each method as specified on lines 4 to 7 of Algorithm 11, and then refining each method by adding parameters as specified by Rule [*InterpreterReturn*-args-intro] and a *stackFrame* variable block as specified by Rule [*InterpreterReturn*-*stackFrame*-intro]. These are added directly to each method's actions, since the parametrised block is moved inside the method by the procedure called on line 9. After this, the *InterpreterNewStackFrame* operations are eliminated from the body of each method by a method `eliminateNewStackFrame()`, because they are moved inside the methods whose references follow them and refined to stack frame initialisation operations.

In the introduce variables step, the `expandWithClassInfo()` method of `CircusAction` applies the rules on lines 3 to 7 of Algorithm 12, which make use of information on the value of *frameClass*. The data refinement on line 13 is performed by `doEFSDataRefinement()`, passing the depth of the operand stack at each point in the method based on the rules in Algorithm 19. Finally, `eliminateVarBlocks()` applies the rules on lines 16 to 27 of Algorithm 12, which eliminate extra variable blocks around various constructs. The removal of the *frameStack* from the state (Algorithm 13) is trivial and has no effect on the method actions so there is nothing to be done for it in our implementation.

The *Circus* model resulting from this stage is extracted as a `String` from the `CThrModel`, using its `toModelString()` method, and written to an output file. The `toModelString()` method of `CThrModel` calls the `toModelString()` method of each `CircusAction` to traverse the *Circus* syntax tree and output the LaTeX representation of each *Circus* construct. If the corresponding C code is desired, the `toCCode()` methods of `CThrModel` and `CircusAction` are used instead to output the C code representation of each *Circus* construct. A C header file is also output containing struct definitions and function prototypes for operations defined in the launcher, to ensure that all the definitions required by the C code are available.

As our prototype is just for the purposes of validating the strategy, we have not performed a direct formal verification of its implementation. However, since we have applied the compilation rules in the implementation in a way that matches the form of the rules in the strategy, which are proved, we are confident of its correctness. The correctness of the implementation is further validated by loading the *Circus* code output from the prototype into CZT to ensure that it is well-formed, and checking the output to ensure it has the expected form.

The well-formedness of the C code output from our prototype is shown by the fact that it compiles without errors or warnings on GCC 7.3.0, using the command `gcc -c -Wall -pedantic`. The choice of warning flags for this compilation matches those used by icecap when launching an icecap program from within Eclipse. We note that our code can only be compiled, and not linked, as creating an SCJVM services implementation to link to our program code is outside the scope of this work.

There have been various considerations raised in producing this prototype. One consideration is that of how to represent the class, field, and method identifiers used in the bytecode. In the model these are represented by given sets, since their representations do not matter provided they can be distinguished from one another and information necessary to the operation of the strategy (specifically, the number of arguments to a method identifier and whether it denotes an instance initialisation method) can be gleaned from them. For simplicity, we just use the identifier strings supplied in the input Java class files, concatenating method and field names with their type signatures, and removing or replacing characters that are not valid in *Circus* identifiers.

Since we apply the compilation rules in our implementation as prescribed in our strategy, we can observe how the individually correct compilation rules fit together. It has highlighted the need to consider the extent of variable blocks. In particular, the loop and conditional introduction rules must match the variable block introduced by the expansion of the `if_icmple` bytecode instruction.

We also found that Rule [resolve-normal-method] must extend the *poppedArgs* variable block to cover the reference to the method action it introduces, in order to match the combination of the

IntepreterNewStackFrame operation and method action reference in Rule [*InterpreterReturn*-args-intro]. In addition, it revealed that the return action must be distributed outside of the variable blocks surrounding conditionals in Rule [conditional-dist]. The form of the methods resulting from the elimination of program counter also made clear the need for *Poll* actions before *Running* in *Started* and *MainThread*, in order to match method calls introduced in the body of methods.

All these considerations have been taken into account in the strategy presented in the previous chapter. In the next section, we discuss some examples whose compilation we have automated using our prototype. We focus on the generated code, and its relation to icecap results.

6.4 Examples

In this section, we evaluate the strategy by considering some examples of SCJ programs. We compare the code generated from the prototype implementation of the compilation strategy to that resulting from the icecap HVM for each of the examples. The examples we have chosen are taken from those developed during the high-integrity Java applications using *Circus* project (which may be found at `www.cs.york.ac.uk/circus/hijac/case.html`).

We particularly focus on SCJ Level 1 examples that illustrate some of the main features of SCJ. These examples cover the full range of bytecode instructions in our subset, and include various examples of loop and conditional constructs to test the strategy. There are three examples we discuss. The first is `PersistentSignal`, discussed in Section 6.4.1, which demonstrates SCJ scheduling behaviour. The second is `Buffer`, discussed in Section 6.4.2, which demonstrates SCJ memory behaviour. Finally, the third example, `Barrier`, which demonstrates a common synchronisation pattern in real-time systems, is discussed in Section 6.4.3.

We have run the examples through both our prototype and the icecap compiler. While running the examples through our prototype, various issues with the compilation strategy and our prototype have been identified and fixed. The first is that the examples make use of bytecode instructions that are not in the representative subset of instructions described in Section 4.3.1. However, since this is a representative subset, the strategy can be easily expanded to handle the missing instructions by analogy to the instructions in the subset. For example binary operation bytecodes can have their semantics defined in a similar way to `iadd`, with compilation rules to handle them similar to those for `iadd` (and their corresponding proofs similar). These extra rules are implemented in our prototype. Conditional instructions can be handled in a similar way to `if_icmple` during bytecode expansion, and subsequently handled by existing compilation rules. Also, while we did not consider `long` values in our strategy, we have implemented handling of operations on `long` values in our prototype, operating on pairs of variables and stack slots.

Array instructions can be represented in programs using classes that contain the individual slots of the array as fields. These fields can then be accessed using methods that select the appropriate field with conditionals over an array index. A full implementation would replace these method calls with specialised communications with the struct manager, which would handle them using C arrays. Although this would require changes to the object/struct manager, it would be simple in terms of the strategy as the structure of the arrays would not change during compilation and very little would need to be performed on the instructions in the interpreter.

We have also found that *poppedArgs* variable blocks around special method calls are not eliminated in Algorithm 12, although variable blocks for normal methods are handled by

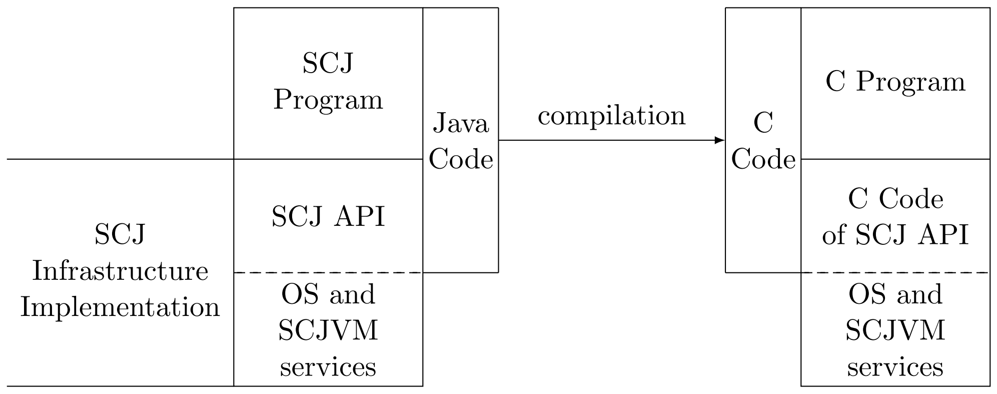


Figure 6.2: The relation of an SCJ program to the SCJ infrastructure in compilation

Rule [*poppedArgs*-elim] and Rule [*poppedArgs*-sync-elim]. Eliminating *poppedArgs* around special methods requires rules similar to these rules to handle each individual special method. The rules required are simple rules to substitute the value of *poppedArgs* into the body of the action and then eliminate the variable block with the initialisation of *poppedArgs*.

After generating the code for each of our examples, we have evaluated the examples by comparing the code generated by our prototype for each of the program methods to that generated by icecap. We focus on the methods of the example programs themselves, rather than the methods of the SCJ API, which are compiled along with the program code. This can be seen in Figure 6.2, which shows the structure of an SCJ program in relation to the infrastructure and compilation.

The SCJ program depends on the SCJ infrastructure implementation, which consists of an SCJ API implementation, possibly written in Java, and the OS and SCJVM services, written in some native language. Only the parts written in Java are subject to compilation, so the OS and SCJVM services are not included in the compilation. How much of the SCJ API implementation is written in Java, and hence included in the code that undergoes compilation, depends upon the OS and SCJVM services. These are generally accessed through native method calls in Java code, but are usually implementation-defined and not visible to end-users, as indicated by the dashed line in Figure 6.2.

In our model, native methods are represented by special methods, which are called using channels, rather than bytecode invoke instructions. Our model of the infrastructure covers the elements in the SCJ standard. In icecap, however, some are implemented in Java, and some are implemented in C. So, when compared to our compilation, icecap deals with more Java code than we do. To account for these differences when passing the examples through the compilation strategy, we provide a small implementation of part of the SCJ API, linking the SCJ code of the examples to the SCJVM via the special methods in our model.

The SCJ API implementation code passed to our prototype is thus different from the SCJ API implementation used by icecap, although the program code is the same, since the methods of the SCJ API are the same and it is only their implementation that differs. We thus, as already mentioned, focus on the program code in our evaluation of the examples. Ensuring the correctness of the API implementation is a separate issue, work on which has begun in [38].

In comparing the methods of the program, we have noted the similarities and differences between our code and the code generated by icecap, and considered why each of the differences is present.

The code used in our comparison can be found in Appendix B.

In what follows, we describe each of the examples and discuss points of our translation to C code that are particularly relevant to each example. In Section 6.4.4, we include a general discussion of the similarities and differences observed while comparing the code generated by our prototype to that generated by icecap for the program methods of each example.

6.4.1 `PersistentSignal`

Our first example is `PersistentSignal`. It consists of a single mission with two event handlers: a periodic handler, `Producer`, and an aperiodic handler, `Worker`. These communicate through an instance of a third class, `PersistentSignal`, after which the example is named. The `PersistentSignal` class contains a boolean flag, with `synchronized` methods to read, set and clear it. `Producer` releases clear the `PersistentSignal` flag and then signal for the `Worker` to release. The `Worker` sets the `PersistentSignal` flag during its release and the `Producer` checks the flag to see if the `Worker` has finished its release. Both the `Producer` and `Worker` produce output to indicate when they are released.

The main purpose of this example is to demonstrate SCJ's scheduling behaviour. The priority of the `Worker` is set higher than the priority of the `Producer`, so the `Worker` always preempts the `Producer`, leaving the flag set at the end of its release before allowing the `Producer` to finish its release. This means that the synchronisation applied to the methods of `PersistentSignal` may not be necessary, but it is good practice due to the possibility of release jitter whereby the scheduler may switch to a thread after a small delay if, for example, the scheduler is running on its own thread or in response to clock interrupts.

The code generated by our prototype for this example is similar to that generated for it by icecap. The operations on local variables and the operand stack are represented by operations on C variables in both the code icecap generates and the code resulting from our strategy. The names of the variables differ between icecap and our implementation, since icecap uses the local variable names from the original SCJ code, which are included in class files for debug purposes, but different names can be used without affecting the correctness of the code.

The synchronisation behaviour is particularly evident in this example, and is handled the same in both the code from icecap and the code from our prototype. The lock is taken just before a call to a synchronized method, and released at the end of the synchronized method. In our model this is represented by the *takeLock* and *releaseLock* channels; icecap uses a `handleMonitorEnterExit()` function to handle both, passing a boolean flag to it to distinguish between taking the lock and releasing the lock. The objects locked on are the same in our code as for icecap: the first argument on the stack when calling the method, and the first local variable when returning from the method.

6.4.2 `Buffer`

Our second example is `Buffer`, which, like the previous example, consists of two event handlers: a periodic handler, `Producer`, and an aperiodic handler, `Consumer`. During a release of `Producer`, it calls the `executeInOuterArea()` method of `ManagedMemory`, passing in an anonymous `Runnable` object stored in a field `_switch` of `Producer`. The `run()` method of the object in `_switch` allocates an instance of `Object` and stores it in a field `data` of `Producer`. Since it is executed via `executeInOuterArea()`, this instance of `Object` is allocated in the memory area outside the per-release memory for Producer, which is the mission memory. The object in data is then stored in a buffer and the Consumer is released, which pops the object from the buffer.

The purpose of this example is to demonstrate the memory behaviour of SCJ. Since the object passed via the buffer is used by both event handlers, it must be allocated in mission memory to ensure that it is available to both event handlers. Since objects are, by default, allocated in an event handler's per-release memory area during its release, this allocation must be performed via executeInOuterArea(). The buffer itself must also be allocated in mission memory, but it does not require use of executeInOuterArea(), since it is allocated during mission initialisation. Since the mission memory is not cleared during the mission, it would eventually run out of space to allocate the objects with repeated releases of Producer. To prevent this, the Producer maintains a count of how many times it has been released and does not allocate the object or store it in the buffer if it has been released more than a set number of times.

Of note is how the use of executeInOuterArea() is represented in the code, since it provides a good example of how method calls are translated. The call to executeInOuterArea() itself is a simple static method call in both the code generated by icecap and the code generated by our strategy, since the SCJ API does not differ between them. Although the implementation of the method differs between the icecap code and the code from our strategy, due to the differing SCJ libraries used, they both contain code to change the memory area, a call to the run() method of the Runnable object passed into the method, and code to change the memory area back to its previous value.

The call to the run() method of the Runnable object is interesting as many classes in the SCJ infrastructure implement Runnable, providing a large set of possible targets for the call. There is a large difference in the set of targets chosen for the method call by icecap and our prototype — the icecap code lists 10 targets, whereas our code lists 4. The only target that appears on both lists is Producer$1, the anonymous class in Producer that is the actual target of the executeInOuterArea() call we are considering. The other three targets in our code are all subclasses of AsyncEventHandler, which is part of the superclass hierarchy for event handlers.

AsyncEventHandler is included in the list of choices for icecap and is selected there by searching the superclasses of the object the method is called on until one of the listed targets is found. In our code, we adopt a different approach, selecting using the object's actual type but directing the call to the class in which the method is defined. This means that while there are three branches of the choice corresponding to subclasses of AsyncEventHandler, the contents of those branches are the same call to AsyncEventHandler's run() method. This is simply a static resolution of the superclass search that icecap conducts, for each of the possible subclasses of AsyncEventHandler in the example program, so it is equivalent. The other targets listed in the icecap code are parts of the SCJ infrastructure that are handled in our model by the *Launcher* and SCJVM services.

6.4.3 Barrier

Our third example is Barrier, which demonstrates a common pattern in real-time systems, where an event only happens when multiple event handlers have signalled their readiness. It is based around a class named Barrier, which implements this pattern. There are three types of event handlers in this example: FireHandler, which is the type of aperiodic event handlers that must signal their their readiness to the Barrier, LaunchHandler, the aperiodic handler that releases when all the `FireHandler`s have signalled their readiness, and `Button`, a periodic handler that simulates events releasing the `FireHandlers`. In the example, two instances of `FireHandler` are created, with corresponding `Button` event handlers: one that releases every 2 seconds, and one that releases every 9 seconds.

When a `FireHandler` releases, it checks if it has already triggered the `Barrier`, and calls a method of the `Barrier` to trigger it if it has not already been triggered, passing a numerical identifier. When the `Barrier` is triggered, it sets a boolean flag corresponding to the passed numerical identifier, then checks if all the boolean flags are set. When all the boolean flags are set, the `Barrier` releases its associated `LaunchHandler` object and resets all the boolean flags. The `LaunchHandler` gives output to indicate when it is released.

This example shows a more complex scheduling behaviour than that of the previous examples. The `FireHandler` and `LaunchHandler` event handlers have higher priority than the `Button` event handlers, so they cannot be interrupted by the periodic events firing. However, the `FireHandler` and `LaunchHandler` handlers have the same priority, and the methods of `Barrier` are synchronized, so the boolean flags in `Barrier` are completely reset before the `LaunchHandler` executes. Due to the differing periods for the `Button` handlers, the first `FireHandler` releases four or five times before the second `FireHandler` releases and `LaunchHandler` executes.

A particular feature of interest in the code generated for this example stems from the fact that it has multiple aperiodic event handler types. This means that a release of an aperiodic handler in our code (as may occur when the `Barrier` is triggered) is represented by a choice between them, although both branches of the choice contain a call to the `release()` method of `AperiodicEventHandler`. The corresponding icecap code simplifies this to a direct call to the `release()` method of `AperiodicEventHandler`, omitting the unnecessary choice. Such a transformation could be made in our strategy, although fully applying it involves eliminating the *getClassIDOf* communication used to get the class identifier used in the choice. As explained in Section 5.7, this requires operating on multiple processes and so we leave it to future work. Note that the fact that there are multiple instances of `Button` and `FireHandler` makes no difference to either the code from icecap or the code from our strategy.

6.4.4 Code comparison

We observed various similarities between our code and that generated by icecap. Firstly, variables are generated to store the contents of stack slots in both, with values being pushed to the stack by assignment to these variables, and operations performed upon them. Local variables of a method are also represented by C variables, and arguments of the method are passed as arguments of the corresponding C function in both our code and icecap's code. There are some differences in the names of variables; we name variables using `var` and a number while icecap uses the name of the variable from Java. In addition, icecap distinguishes stack slots for different types, although the basic approach is the same.

Method calls also display similarities, particularly for non-virtual method calls, which are simple C function calls in both our code and icecap's code. Virtual method calls display some differences in how the method to be called is selected (discussed below), but the method call itself is as in the non-virtual case. Calls to `synchronized` methods are also compiled in the same way, with the lock being taken on an object before the method is called and released just before the method ends, where the operations of taking and releasing locks are performed by calls to infrastructure functions.

We have also observed that field accesses are the same in both our code and in icecap. The fields are accessed in our code by casting a variable storing the pointer to the object, first to `uintptr_t` to ensure it is expanded correctly on systems where pointers are wider than 32 bits (since a single variable is a 32-bit integer in our code), and then to a pointer to the struct type for the object's class, and by finally accessing the field via a C field access. The icecap code performs the access in the same way, although an intermediate variable is used, a custom `pointer` type is used in place of `uintptr_t`, a few other pointer casts are applied before the final cast to the class struct, and an optional memory offset is allowed for. The intermediate variable does not affect the semantics of the access, nor do the additional pointer casts, and it is not clear why they are present in the icecap code since they are not necessary. The `pointer` type is defined to be equivalent to the C99 `uintptr_t` type (although it handles those compilers that may not support C99), so there is no difference in the semantics of our code and the icecap code on this point. For the memory offset, we assume memory addresses used by the memory manager are small enough to fit in a 32-bit JVM word. Future work could add an offset to handle heaps outside that range if necessary.

There are various differences between our code and icecap; many of these relate to areas we have explicitly not considered in our strategy. In other cases, we have chosen to diverge from icecap's approach. These differences are discussed next.

Firstly, there are several methods in our code that do not have corresponding methods in the code generated by icecap. This is due to a combination of different factors. Some methods are not present in the code input to icecap code due to differences in the version of the SCJ API used by our code and icecap. Other methods are present in the code input to both our prototype and icecap, yet have no C code generated for them in icecap. The lack of code for these methods in icecap is due to a difference between our prototype and icecap in how the set of methods to be compiled is computed. Our prototype generates C code for all methods passed to it, whereas icecap computes which methods are required for the program, beginning from the main method that forms the starting point of the launcher. While this does exclude one method (`main_BoundedBuffer_isFull`) that is defined in the example code but not used, the other methods have no corresponding icecap code due to the fact that they appear not to be called in icecap's launcher infrastructure. This would appear to be a deficiency in icecap's implementation of the SCJ startup procedure, where fixed sizes are used for the immortal and mission memory rather than obtaining them from the `Safelet` and `Mission` provided by the program.

Another difference is that the icecap code passes a frame pointer, `fp`, to each function and defines a stack pointer variable, `sp`, in each function. These are used to manage a stack, which is used in addition to the stack slot variables. This stack allows the compiled code to interact with interpreted code, since the interpreted code uses this stack rather than having predefined stack slots (which are computed during the compilation process). We do not require this feature in our code, since all our code is compiled. For the same reason, we also do not generate the code to swap stack slot variables to and from this stack. There are also some infrastructure methods in icecap that accept their arguments using the stack, and a few of the generated functions in the icecap code (such as `main_BoundedBuffer_init`) pop their arguments from the stack rather than taking them as function arguments. This is, of course, unnecessary for our code, where we adopt the same approach of passing arguments as C function arguments for all methods.

The icecap code also uses a different approach for returning values from functions. In the icecap code, return values are passed using the stack passed into the function, with the return value popped from the stack in the calling function. In our code, we do not pass a stack pointer, so pointers to the stack slot variables in which the return values are to be placed are passed instead. This approach used in our code is preferable to using C return values as it scales better to `long` values, which require two variables. We note that icecap functions returning small values, particularly `boolean` values in the case of our examples, instead use the `int16` value returned from each function (normally used to signal exceptions in the icecap code) to pass the return value. This is a somewhat inconsistent approach to passing the return values, but it perhaps makes best use of space for small values.

There is a lot of exception handling code in icecap that is not present in our code, since we do not handle exceptions. As we have already seen, the return values for signalling exceptions are also used to pass small return values of the method. We omit the return values completely in our code, since we do not need to signal exceptions, but our system of passing method return values frees up the function return value for use as part of an exception handling system in future work.

There is also a difference in how control flow constructs are compiled. Jumps in the bytecode are translated by icecap using `goto` statements in C. This allows bytecode instructions to be translated more directly, but it means the resulting code is not fully MISRA-C compliant. In our compilation strategy, we avoid the use of `goto` by considering the control-flow graph of each method and introducing structures such as loops and conditionals. This has the added advantage of making the resulting code more readable and, since we have have certainty that this transformation does not change the semantics of the code, it is not necessary to use the most direct translation as icecap does.

Differences in the code arising from the different API provided for our code versus icecap also show themselves in the generated code. Array operations in icecap are translated using C array accesses but, since we do not have arrays, we model them as objects. It is expected that future work that adds handling of arrays to our compilation strategy would produce code similar to that of icecap. There are also some places where `static` fields holding constant values for memory sizes are additionally declared `final` in our code, where in icecap they merely hold the same value as a `final` field. This means accesses to these fields appear in the icecap code, but the values of the fields are inlined in our code. However, from `static` field accesses in our SCJ API implementation (such as in `devices_Console_read` for each of the examples), we see that they are translated in the same way in our code as in icecap: by an access to a field of a global struct containing class fields.

Finally, there is also a difference in how virtual method call targets are chosen in our code versus that of icecap. In icecap, the superclasses of the target object are searched to determine which method should be executed, whereas in our code a choice is made over the class of the target object, with the superclasses searched at compile time. Our approach means the work of searching for the class containing the definition need not be performed at runtime, but results in several conditional branches with the same body for classes that have a common superclass. As a future optimisation in our code, such branches could be merged and a switch statement could be used for a more efficient choice over classes. The icecap code also removes the search entirely if there is only a single possible target. Such a transformation could be made in our strategy, although fully applying it involves eliminating the *getClassIDOf* communication used to get the class identifier used in the choice. As explained in Section 5.7, this requires operating on multiple processes and so we leave it to future work. In any case, the different approaches to selecting the target method yield the same target at run-time.

We also noted a difference in the size of the code generated by our prototype versus that generated by icecap. Our prototype generates two files for each example: a `.c` file containing the code for each of the methods, and a `.h` file containing struct definitions and prototypes for infrastructure functions (the provision of which is outside the scope of this thesis). The files generated by icecap include many pre-defined files, that do not result from compilation of the examples. Namely, the files that are generated by icecap from the code of each of the examples are a `methods.c` file, containing the code of each method, a `methods.h` file, defining constants used to identify each of the methods, a `classes.c` file, defining variables containing class information, and a `classes.h` file, defining struct types for the objects of each class. The `methods.c` file corresponds to the `.c` file generated by our prototype, and the `classes.h` file corresponds to the `.h` file generated by our prototype. So we compare the sizes of those files. The class information in `classes.c` and the method identifiers in `methods.h` are included in icecap to support interpretation of bytecode, which is not necessary in our code, so we do not include these files in our comparison.

The sizes of the `.c` files generated by our prototype are: 2576 lines for `PersistentSignal`, 2748 lines for `Buffer`, and 2787 lines for `Barrier`. The sizes of the corresponding `methods.c` files generated by icecap are 63968 lines for `PersistentSignal`, 65383 lines for `Buffer`, and 64619 lines for `Barrier`. The sizes of the `.h` files generated by our prototype are: 542 lines for `PersistentSignal`, 560 lines for `Buffer`, and 551 lines for `Barrier`. The sizes of the corresponding `classes.h` files generated by icecap are 1164 lines for `PersistentSignal`, 1187 lines for `Buffer`, and 1181 lines for `Barrier`. Note that these sizes are the size of the complete files, including blank lines and comments.

The icecap files are clearly much larger, but this includes the larger SCJ API implementation of icecap. Extracting the definitions of each of the program methods from the `.c` files generated by the prototype gives 362 lines for `PersistentSignal`, 508 lines for `Buffer`, and 547 lines for `Barrier`. Similarly, extracting the program method definitions from the `methods.c` files generated by icecap gives 1634 lines for `PersistentSignal`, 2041 lines for `Buffer`, and 2241 lines for `Barrier`. The difference in size for the program methods is smaller, but the size of the icecap method code is still more than four times the size of the code generated by our prototype for each example. This is, however, largely accounted for by the fact that icecap includes extra code for exception handling and comments indicating which line of the original Java code each line of C code corresponds to. For both our prototype and icecap, the C code is longer than the original Java files, the sizes of which are: 270 lines for `PersistentSignal`, 343 lines for `Buffer`, and 318 lines for `Barrier`. The input to icecap also includes an additional 10-line file containing a `main()` method that invokes the icecap launcher infrastructure, and may be taken as part of the *Launcher* in our model. The longer size for the C code over the Java code follows from the fact that each line of Java code may be translated by multiple bytecode instructions.

Overall, our code is similar to that of icecap; differences are justified in that they are more suited to the particular approach we adopt: not interpreting and ensuring MISRA-C compliance of the code. This thus provides additional confidence in the validity of the code generated by our strategy.

6.5 Final Considerations

In this chapter, we have considered various ways in which our model and compilation strategy can be evaluated, and their correctness validated. The models used as input to the strategy have been validated by using CZT to perform syntax and type checking, and performing some proofs using Z/EVES on the schemas defining instruction semantics. We have seen that this ensures that the model is well-formed and provides a means to deduce the preconditions that must be satisfied for each bytecode instruction. The preconditions found match those checked by JVM bytecode verification, ensuring our semantics is correct for standard Java bytecode.

We have also discussed the proofs of the compilation rules and the source of the laws used used to prove those rules, seeing that the algebraic proof style of the rules gives great certainty of the proof's correctness by formally justifying each step in the proof. As we mentioned, the laws we have used come from existing *Circus* laws taken from various sources and laws we have proved in Isabelle/UTP. This basis of laws known to be correct provides further assurance of the correctness of the proofs.

Finally, we have discussed our prototype implementation of the compilation strategy and the assurance that may be gained from considering some examples. The tool shows how the individual compilation rules fit together as a complete whole, allowing us to check how the rules act as part of the strategy. The examples we have considered show that the code we generate is generally comparable to that generated by icecap. The few differences observed between our code and icecap's code arise from design choices that enhance the generated code.

While we have used the prototype to check the form of the generated code, it can also give us an idea of the complexity of the strategy so that we can judge how viable it would be were we to make a full implementation of it. By packaging the prototype as a `.jar` file we can execute it from a command line and use the `time` command to measure its execution time. We have performed this in Ubuntu 18.04 running on an Intel Core i5-520M processor. Averaging wall-clock time across 10 runs of the prototype for each of our examples yields 2.50 seconds for `PersistentSignal`, 2.78 seconds for `Barrier`, and 2.67 seconds for `Buffer`.

From the output of the prototype indicating which stages of the strategy are executing, the bulk of the time appears to be spent in introducing sequential composition. More detailed tracing of the time taken shows that this is due to the fact that the control flow graph is reconstructed after each compilation rule is applied. The number of reachable nodes is very high at the start of the compilation strategy, but reduces by a large amount during sequential composition introduction, since most of the edges between nodes represent sequential composition. This time could be reduced by using a more sophisticated strategy to perform local updates of the control flow graph, potentially reducing the execution time of the prototype by up to 2 seconds. With other optimisations, such as more efficient data structures and pattern matching strategies, this could give reasonable execution time, even for large programs.

It is difficult to produce similar measurements of compilation time for icecap, since icecap is designed as an Eclipse plugin and cannot be separated from Eclipse to allow measurement of compilation time. However, we note that the compilation time for our examples in icecap seems to be of the same order of magnitude as for our prototype.

Our implementation is just a prototype and so any measurements of its running time are only approximations of the efficiency of our compilation strategy. It is more helpful to consider the asymptotic complexity of the compilation strategy, to determine if it scales well in an optimised implementation. Assuming an input program consists of m methods containing an average of n instructions each, and that the local updates of the control flow graph are made to take constant time, then the time complexity of our strategy is at most $\mathcal{O}(m^3 n)$. This is because, firstly, the loop on line 3 of Algorithm 1 may loop once for each of the m methods if only one method is separated in each iteration. At least one method will be separated in each iteration and it is expected that more than one will be separated in most iterations, so m is a conservative upper bound on the number of iterations in the loop. Within the loop, Algorithm 6 checks each method call instruction, of which there may be up to mn, and for each target of the method call, of which there could be as many as there are methods, m, searches its superclasses for an implementation, of which there may be as many as there are methods, m.

None of the other algorithms contributes as much as Algorithm 6 to the time complexity, since all the other elimination of program counter algorithms simply iterate over each node in at most one loop. Even Algorithm 3, which has two nested loops, is linear in the number of instructions, since any additional iteration of the inner loop means a sequential composition is introduced so the nodes are merged. Thus there is one fewer node and one fewer iteration of the outer loop.

The elimination of frame stack algorithms are generally at most $\mathcal{O}(m^2 n)$, since they may transform each call of the potential mn calls to the m methods. The data refinement of objects has a separate complexity, since it is determined by the number of classes and fields. It is unlikely to contribute more to complexity than the instructions, since each field should be accessed by at least one instruction.

The overall complexity is thus $\mathcal{O}(m^3 n)$, but this is an upper bound and in most cases iteration will not be over all methods. It may be possible to find iteration strategies to reduce this asymptotic complexity by iterating over the methods fewer times.

All these considerations serve to validate the correctness of the model and strategy, and shows that our strategy is a promising basis for a correct-by-construction ahead-of-time SCJ-to-C compiler.

Chapter 7

Conclusions

In this chapter we conclude by summarising the contributions of this dissertation in Section 7.1. We then discuss directions of future work in Section 7.2.

7.1 Summary of Contributions

We have considered the safety-critical variant of Java, in Section 2.3, and the virtual machines designed to run programs written in it, in Section 2.4. None of the virtual machines is formally verified and many of them precompile programs to native code. Given the need for a formally verified virtual machine, we have developed a framework within which an SCJVM can be verified.

Having noted that SCJ virtual machines employ compilation, we have surveyed some of the work on compiler correctness, particularly those related to Java compilation, in Section 2.5. We have established that two approaches to compiler correctness have been used: the commuting-diagram approach and the algebraic approach. We have adopted the algebraic approach and chosen *Circus*, described in Section 2.6, as a specification language.

To specify an SCJVM we have identified the requirements of the virtual machine services to support SCJ programs. We have also constructed a formal model of those requirements in the *Circus* specification language. These virtual machine services requirements and their formal model are discussed in Chapter 3.

Contact with one of the authors of the SCJ specification has allowed us to obtain clarifications where the specification was unclear. The development of the formal model has helped in the identification of the areas that require clarification. It may be noted that the interface we have defined is not the only one that can support SCJ, but its overall functionality must be present in all SCJ virtual machines in some way. The SCJ specification has been changed to reflect many of these clarifications. In particular, the current thread during an interrupt, the backing store space required during mission setup, and the initialisation of the `Safelet` have all been clarified as a result of our contact with the authors of the SCJ specification.

We have also created a formal model of the core execution environment that executes SCJ programs in an SCJVM. This model has been created by identifying a minimal subset of Java bytecode and defining its semantics, and then constructing a *Circus* model of an interpreter for that subset with the necessary infrastructure around it. We have discussed the core execution environment model in Chapter 4.

Finally, we have specified a strategy for compiling SCJ bytecode to C, presented in Chapter 5. This is a strategy to apply individual compilation rules, which are stated as algebraic laws, to transform our model of an SCJVM interpreter, loaded with an SCJ bytecode program, to a *Circus* representation of the equivalent C code. Since the compilation rules are stated formally as *Circus* refinement laws, we can, and have, written proofs of them. We have discussed these proofs in Section 6.2. This allows us to be sure of their correctness, so that our compilation strategy preserves the semantics of the original SCJ bytecode. In this way, we have created a strategy for correct compilation from SCJ bytecode to C.

Our work is done in the context of a wider effort to facilitate fully verified SCJ programs. There has already been work on generating correct SCJ programs from *Circus* specifications [25, 26] and formalisation of the SCJ memory model [23]. These works allow for verification of SCJ programs, with our work covering the next stage in ensuring those programs can be run correctly.

Since our work addresses the execution of Java bytecode, it must still be ensured that SCJ programs can be compiled to bytecode correctly. Since SCJ does not make any syntactic changes to Java and the semantic changes can be dealt with at the level of Java bytecode, a standard Java compiler suffices for SCJ. As discussed earlier, there has been plenty of work on correct compilation of Java programs [34, 54, 65, 112, 113] so it can be seen that there is already sufficient work to permit correct compilation to Java bytecode. This then leaves us with correct SCJ programs in Java bytecode and the focus of our work is on the next stage of running those programs.

Finally, as we are adopting the approach of compilation to C, it must also be ensured that the C code can be compiled correctly. We note that there has been much work on verified C compilation [18, 57, 59–61] and, in particular, that the CompCert project provides a functioning formally verified C compiler that can be used.

So, our work provides the basis for the implementation of a verified compiler, the development of which would provide the final piece required for complete verification of SCJ programs down to executable code.

7.2 Future Work

There are various possibilities for future work arising from our work. Firstly, our work may be further validated by consideration of a wider range of examples. This may involve further extension of the model and compilation strategy to consider instructions and features not covered by our work. These extensions would not involve significant changes to the strategy, since most of the instructions not included in our subset are similar to those in our subset.

A further direction for future work to validate the strategy would be to mechanise the compilation rules and their proofs using an automated theorem prover, such as Isabelle/UTP. This would confirm the correctness of the rules and allow for easier reasoning about the strategy as whole. Code generation from such a mechanisation could also be used to produce an implementation of the strategy.

Our strategy also shows how the algebraic approach developed in [101] may be adapted to compile from low-level languages to higher-level languages. Future work could build upon this to develop compilation strategies for other low-level languages in a similar way, contributing to wider work on the algebraic approach to compilation.

Other possible directions for future work include the full verification of an SCJ virtual machine using our framework or even the creation of a correct-by-construction virtual machine from our specification. The option of deriving a correct virtual machine from our specification may be more desirable than verifying an existing one. This is because virtual machines can often be complex and therefore difficult to verify in a structured way. Moreover, while the effort of proving a virtual machine correct may uncover bugs, it may be a challenge to fix them. Also, the design of an existing virtual machine may not exactly meet the structure of our specification, requiring restructuring to allow the proof effort to begin. The work verifying the icecap scheduler in [38] shows this, since the tight coupling between components in icecap made modelling and verification challenging.

On the other hand, the fact that *Circus* allows for refinement means that a correct virtual machine can be constructed from our model in a stepwise and modular fashion, being shown to be correct at each stage of the process. Facilitating such work is the ultimate aim of our work, in order to provide for the correct running of SCJ programs.

Appendix A

Compilation Rules

This appendix contains the compilation rules used in the compilation strategy described in Chapter 5. We present the compilation rules in sections corresponding to the sections of Chapter 5 in which they are first used. The rules for Section 5.3.2 are presented in Section A.1.1, the rules for Section 5.3.3 in Section A.1.2, the rules for Section 5.3.4 in Section A.1.3, the rules for Section 5.3.5 in Section A.1.4, the rules for Section 5.3.6 in Section A.1.5, the rules for Section 5.4.1 in Section A.2.1, the rules for Section 5.4.2 in Section A.2.2, the rules for Section 5.4.3 in Section A.2.3, and the rules for Section 5.5 in Section A.3.

We also include in this appendix additional algorithms referenced in the compilation strategy but not included in the main body in the thesis. They are included in the sections of this appendix corresponding to the sections of Chapter 5 in which they are used.

Finally, we also list the algebraic laws applied by the algorithms of our compilation strategy in Section A.4 at the end of this appendix.

A.1 Elimination of Program Counter

A.1.1 Expand Bytecode

Rule [*pc*-expansion]**.** Given $bc : \mathit{ProgramAddress} \pfun \mathit{Bytecode}$,

$$\mathit{HandleInstruction}_{bc} = \mathbf{if}\ \Box_{i \in \mathrm{dom}\, bc}\ pc = i \longrightarrow \mathit{HandleInstruction}_{bc}\ \mathbf{fi}$$

Rule [*HandleInstruction*-refinement]**.** Given $i : \mathit{ProgramAddress}$, if $i \in \mathrm{dom}\, bc$ then,

$$\begin{array}{l} \mathbf{if}\ \cdots \\ \Box\ pc = i \longrightarrow \mathit{HandleInstruction}_{bc} \\ \cdots \\ \mathbf{fi} \end{array} \sqsubseteq_A \begin{array}{l} \mathbf{if}\ \cdots \\ \Box\ pc = i \longrightarrow \mathit{handleAction}(bc\, i) \\ \cdots \\ \mathbf{fi} \end{array}$$

where *handleAction* is a syntactic function defined by Table 5.1.

Rule [*CheckSynchronizedReturn*-sync-refinement]**.** Given $i : \mathit{ProgramAddress}$,

$$
\begin{array}{l}
\mathbf{if}\ \cdots \\
\ [\!]\ pc = i \longrightarrow \\
\quad \mathit{CheckSynchronizedReturn}\ ;\ A \\
\cdots \\
\mathbf{fi}
\end{array}
\ \sqsubseteq_A\
\begin{array}{l}
\mathbf{if}\ \cdots \\
\ [\!]\ pc = i \longrightarrow \\
\quad \mathit{releaseLock}!((\mathit{last\,frameStack}).\mathit{localVariables}\ 1) \\
\quad \longrightarrow \mathit{releaseLockRet} \longrightarrow \mathbf{Skip}\ ;\ A \\
\cdots \\
\mathbf{fi}
\end{array}
$$

provided

$$
\begin{array}{l}
\exists\, c : \mathit{Class};\ m : \mathit{MethodID} \mid \\
\quad c \in \operatorname{ran} cs \land m \in \operatorname{dom} c.\mathit{methodEntry} \bullet \\
\quad i \in c.\mathit{methodEntry}\ m \mathrel{..} c.\mathit{methodEnd}\ m \land \\
\quad m \in c.\mathit{synchronizedMethods} \land m \notin c.\mathit{staticMethods}
\end{array}
$$

Rule [*CheckSynchronizedReturn*-nonsync-refinement]**.** Given $i : \mathit{ProgramAddress}$,

$$
\begin{array}{l}
\mathbf{if}\ \cdots \\
\ [\!]\ pc = i \longrightarrow \mathit{CheckSynchronizedReturn}\ ;\ A \\
\cdots \\
\mathbf{fi}
\end{array}
\ \sqsubseteq_A\
\begin{array}{l}
\mathbf{if}\ \cdots \\
\ [\!]\ pc = i \longrightarrow A \\
\cdots \\
\mathbf{fi}
\end{array}
$$

provided

$$
\begin{array}{l}
\exists\, c : \mathit{Class};\ m : \mathit{MethodID} \mid \\
\quad c \in \operatorname{ran} cs \land m \in \operatorname{dom} c.\mathit{methodEntry} \bullet \\
\quad i \in c.\mathit{methodEntry}\ m \mathrel{..} c.\mathit{methodEnd}\ m \land \\
\quad m \notin c.\mathit{synchronizedMethods} \lor m \in c.\mathit{staticMethods}
\end{array}
$$

A.1.2 Introduce Sequential Composition

Rule [sequence-intro]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$ and

$$
\begin{array}{l}
\{\mathit{frameStack} \neq \varnothing\}\ ;\ A \\
= \\
\{\mathit{frameStack} \neq \varnothing\}\ ;\ A\ ;\ \{\mathit{frameStack} \neq \varnothing\}
\end{array}
$$

then,

$$
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad\ [\!]\ \mathit{frameStack} \neq \varnothing \longrightarrow \\
\quad\quad \mathbf{if}\ \cdots \\
\quad\quad\ [\!]\ pc = i \longrightarrow A\ ;\ pc := j \\
\quad\quad \cdots \\
\quad\quad\ [\!]\ pc = j \longrightarrow B \\
\quad\quad \cdots \\
\quad\quad \mathbf{fi}\ ;\ \mathit{Poll}\ ;\ X \\
\quad \mathbf{fi}
\end{array}
\ \sqsubseteq_A\
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad\ [\!]\ \mathit{frameStack} \neq \varnothing \longrightarrow \\
\quad\quad \mathbf{if}\ \cdots \\
\quad\quad\ [\!]\ pc = i \longrightarrow \\
\quad\quad\quad A\ ;\ pc := j\ ;\ \mathit{Poll}\ ;\ B \\
\quad\quad \cdots \\
\quad\quad\ [\!]\ pc = j \longrightarrow B \\
\quad\quad \cdots \\
\quad\quad \mathbf{fi}\ ;\ \mathit{Poll}\ ;\ X \\
\quad \mathbf{fi}
\end{array}
$$

A.1.3 Introduce Loops and Conditionals

Rule [`if`-conditional-intro]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$, $i \neq k$, and

$$
\begin{array}{l}
\{\mathit{frameStack} \neq \varnothing\} \,;\; A \,;\; P \\
= \\
\{\mathit{frameStack} \neq \varnothing\} \,;\; A \,;\; P \,;\; \{\mathit{frameStack} \neq \varnothing\}
\end{array}
$$

then

$$
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad \,\square\, \mathit{frameStack} \neq \varnothing \longrightarrow \\
\quad\quad \mathbf{if} \cdots \\
\quad\quad \,\square\, pc = i \longrightarrow A; \\
\quad\quad\quad (\mathbf{var}\, \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P; \\
\quad\quad\quad pc := \mathbf{if}\, b\, \mathbf{then}\, j\, \mathbf{else}\, k) \\
\quad\quad \cdots \\
\quad\quad \,\square\, pc = k \longrightarrow B \,;\; pc := j \\
\quad\quad \cdots \\
\quad\quad \mathbf{fi} \,;\, \mathit{Poll} \,;\; X \\
\quad \mathbf{fi}
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad \,\square\, \mathit{frameStack} \neq \varnothing \longrightarrow \\
\quad\quad \mathbf{if} \cdots \\
\quad\quad \,\square\, pc = i \longrightarrow A; \\
\quad\quad\quad (\mathbf{var}\, \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P; \\
\quad\quad\quad \mathbf{if}\, b \longrightarrow \mathbf{Skip} \\
\quad\quad\quad \,\square\, \neg\, b \longrightarrow pc := k \,;\; \mathit{Poll} \,;\; B \\
\quad\quad\quad \mathbf{fi}) \,;\; pc := j \\
\quad\quad \cdots \\
\quad\quad \,\square\, pc = k \longrightarrow B \,;\; pc := j \\
\quad\quad \cdots \\
\quad\quad \mathbf{fi} \,;\, \mathit{Poll} \,;\; X \\
\quad \mathbf{fi}
\end{array}
$$

Rule [`if-else`-conditional-intro]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$, $i \neq k$, and

$$
\begin{array}{l}
\{\mathit{frameStack} \neq \varnothing\} \,;\; A \,;\; P \\
= \\
\{\mathit{frameStack} \neq \varnothing\} \,;\; A \,;\; P \,;\; \{\mathit{frameStack} \neq \varnothing\}
\end{array}
$$

then

$$
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad \,\square\, \mathit{frameStack} \neq \varnothing \longrightarrow \\
\quad\quad \mathbf{if} \cdots \\
\quad\quad \,\square\, pc = i \longrightarrow A; \\
\quad\quad\quad (\mathbf{var}\, \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P; \\
\quad\quad\quad pc := \mathbf{if}\, b\, \mathbf{then}\, j\, \mathbf{else}\, k) \\
\quad\quad \cdots \\
\quad\quad \,\square\, pc = j \longrightarrow B \,;\; pc := x \\
\quad\quad \cdots \\
\quad\quad \,\square\, pc = k \longrightarrow C \,;\; pc := x \\
\quad\quad \cdots \\
\quad\quad \mathbf{fi} \,;\, \mathit{Poll} \,;\; X \\
\quad \mathbf{fi}
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad \,\square\, \mathit{frameStack} \neq \varnothing \longrightarrow \\
\quad\quad \mathbf{if} \cdots \\
\quad\quad \,\square\, pc = i \longrightarrow A; \\
\quad\quad\quad (\mathbf{var}\, \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P; \\
\quad\quad\quad \mathbf{if}\, b \longrightarrow pc := j \,;\; \mathit{Poll} \,;\; B \\
\quad\quad\quad \,\square\, \neg\, b \longrightarrow pc := k \,;\; \mathit{Poll} \,;\; C \\
\quad\quad\quad \mathbf{fi}) \,;\; pc := x \\
\quad\quad \cdots \\
\quad\quad \,\square\, pc = j \longrightarrow B \,;\; pc := x \\
\quad\quad \cdots \\
\quad\quad \,\square\, pc = k \longrightarrow C \,;\; pc := x \\
\quad\quad \cdots \\
\quad\quad \mathbf{fi} \,;\, \mathit{Poll} \,;\; X \\
\quad \mathbf{fi}
\end{array}
$$

Rule [conditional-intro]**.** Given $i : \mathit{ProgramAddress}$, if $i \neq j$, $i \neq k$, and

$$\begin{array}{l}\{\mathit{frameStack} \neq \varnothing\} \; ; \; A \; ; \; P \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \; ; \; A \; ; \; P \; ; \; \{\mathit{frameStack} \neq \varnothing\}\end{array}$$

then

```
$\mu X \bullet$
    $\mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip}$
    $[\!]\ \mathit{frameStack} \neq \varnothing \longrightarrow$
        $\mathbf{if} \cdots$
        $[\!]\ pc = i \longrightarrow A;$
            $(\mathbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            $pc := \mathbf{if}\ b\ \mathbf{then}\ j\ \mathbf{else}\ k)$
        $\cdots$
        $[\!]\ pc = j \longrightarrow B$
        $\cdots$
        $[\!]\ pc = k \longrightarrow C$
        $\cdots$
        $\mathbf{fi}\ ;\ \mathit{Poll}\ ;\ X$
    $\mathbf{fi}$
```

$\sqsubseteq_A$

```
$\mu X \bullet$
    $\mathbf{if}\ \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip}$
    $[\!]\ \mathit{frameStack} \neq \varnothing \longrightarrow$
        $\mathbf{if} \cdots$
        $[\!]\ pc = i \longrightarrow A;$
            $(\mathbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            $\mathbf{if}\ b \longrightarrow pc := j\ ;\ \mathit{Poll}\ ;\ B$
            $[\!]\ \neg\, b \longrightarrow pc := k\ ;\ \mathit{Poll}\ ;\ C$
            $\mathbf{fi})$
        $\cdots$
        $[\!]\ pc = j \longrightarrow B$
        $\cdots$
        $[\!]\ pc = k \longrightarrow C$
        $\cdots$
        $\mathbf{fi}\ ;\ \mathit{Poll}\ ;\ X$
    $\mathbf{fi}$
```

Rule [`while`-loop-intro1]. Given $i : \mathit{ProgramAddress}$, if $i \neq j$,

$$\{\mathit{frameStack} \neq \varnothing\} \;;\; A \;;\; P \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \;;\; A \;;\; P \;;\; \{\mathit{frameStack} \neq \varnothing\}$$

and

$$\{\mathit{frameStack} \neq \varnothing\} \;;\; C \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \;;\; C \;;\; \{\mathit{frameStack} \neq \varnothing\}$$

then

```
$\mu X \bullet$
    **if** $\mathit{frameStack} = \varnothing \longrightarrow$ **Skip**
    $[]\ \mathit{frameStack} \neq \varnothing \longrightarrow$
        **if** $\cdots$
        $[]\ pc = i \longrightarrow A;$
            $($**var** $\mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            $pc :=$ **if** $b$ **then** $j$ **else** $k)$
        $\ldots$
        $[]\ pc = j \longrightarrow B$
        $\ldots$
        $[]\ pc = k \longrightarrow C \;;\; pc := i$
        $\ldots$
        **fi** $;\ \mathit{Poll} \;;\; X$
    **fi**
```

$$\sqsubseteq_A$$

```
$\mu X \bullet$
    **if** $\mathit{frameStack} = \varnothing \longrightarrow$ **Skip**
    $[]\ \mathit{frameStack} \neq \varnothing \longrightarrow$
        **if** $\cdots$
        $[]\ pc = i \longrightarrow (\mu Y \bullet A;$
            $($**var** $\mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            **if** $b \longrightarrow$ **Skip**
            $[]\ \neg\ b \longrightarrow$
                $pc := k \;;\; \mathit{Poll} \;;\; C;$
                $pc := i \;;\; \mathit{Poll} \;;\; Y$
            **fi**$)) \;;\; pc := j$
        $\ldots$
        $[]\ pc = j \longrightarrow B$
        $\ldots$
        $[]\ pc = k \longrightarrow C \;;\; pc := i$
        $\ldots$
        **fi** $;\ \mathit{Poll} \;;\; X$
    **fi**
```

Rule [`while`-loop-intro2]. Given $i : \mathit{ProgramAddress}$, if $i \neq j$,

$$\{\mathit{frameStack} \neq \varnothing\} \; ; \; A \; ; \; P$$
$$=$$
$$\{\mathit{frameStack} \neq \varnothing\} \; ; \; A \; ; \; P \; ; \; \{\mathit{frameStack} \neq \varnothing\}$$

and

$$\{\mathit{frameStack} \neq \varnothing\} \; ; \; B;$$
$$=$$
$$\{\mathit{frameStack} \neq \varnothing\} \; ; \; B \; ; \; \{\mathit{frameStack} \neq \varnothing\}$$

then

```
$\mu X \bullet$
    if $\mathit{frameStack} = \varnothing \longrightarrow$ Skip
    $[\!]$ $\mathit{frameStack} \neq \varnothing \longrightarrow$
        if $\cdots$
        $[\!]$ $pc = i \longrightarrow A;$
            $(\textbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            $pc := \textbf{if}\ b\ \textbf{then}\ j\ \textbf{else}\ k)$
        $\ldots$
        $[\!]$ $pc = j \longrightarrow B \; ; \; pc := i$
        $\ldots$
        $[\!]$ $pc = k \longrightarrow C$
        $\ldots$
        fi ; $\mathit{Poll}$ ; $X$
    fi
```

$$\sqsubseteq_A$$

```
$\mu X \bullet$
    if $\mathit{frameStack} = \varnothing \longrightarrow$ Skip
    $[\!]$ $\mathit{frameStack} \neq \varnothing \longrightarrow$
        if $\cdots$
        $[\!]$ $pc = i \longrightarrow (\mu Y \bullet A;$
            $(\textbf{var}\ \mathit{value}1, \mathit{value}2 : \mathit{Word} \bullet P;$
            if $b \longrightarrow$
                $pc := j \; ; \; \mathit{Poll} \; ; \; B;$
                $pc := i \; ; \; \mathit{Poll} \; ; \; Y$
            $[\!]$ $\neg\, b \longrightarrow$ Skip
            fi)) ; $pc := k$
        $\ldots$
        $[\!]$ $pc = j \longrightarrow B \; ; \; pc := i$
        $\ldots$
        $[\!]$ $pc = k \longrightarrow C$
        $\ldots$
        fi ; $\mathit{Poll}$ ; $X$
    fi
```

Rule [`do-while`-loop-intro]. Given $i : \mathit{ProgramAddress}$, if $i \neq j$,

$$\begin{array}{l}\{\mathit{frameStack} \neq \varnothing\} \mathrel{;}\ A \mathrel{;}\ P \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \mathrel{;}\ A \mathrel{;}\ P \mathrel{;}\ \{\mathit{frameStack} \neq \varnothing\}\end{array}$$

then

```
μX •
    if frameStack = ∅ ⟶ Skip
    ▯ frameStack ≠ ∅ ⟶
        if ···
        ▯ pc = i ⟶ A;
            (var value1, value2 : Word • P;
            pc := if b then i else j)
        ...
        ▯ pc = j ⟶ B
        ...
        fi ; Poll ; X
    fi
```

$\sqsubseteq_A$

```
μX •
    if frameStack = ∅ ⟶ Skip
    ▯ frameStack ≠ ∅ ⟶
        if ···
        ▯ pc = i ⟶ (μY • A
            (var value1, value2 : Word • P;
            if b ⟶ pc := i ; Poll ; Y
            ▯ ¬ b ⟶ Skip
            fi)) ; pc := j
        ...
        fi ; Poll ; X
    fi
```

Rule [infinite-loop-intro]. Given $i : \mathit{ProgramAddress}$, if

$$\begin{array}{l}\{\mathit{frameStack} \neq \varnothing\} \mathrel{;}\ A \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \mathrel{;}\ A \mathrel{;}\ \{\mathit{frameStack} \neq \varnothing\}\end{array}$$

then

```
μX •
    if frameStack = ∅ ⟶ Skip
    ▯ frameStack ≠ ∅ ⟶
        if ···
        ▯ pc = i ⟶
            A ; pc := i
        ...
        fi ; Poll ; X
    fi
```

$\sqsubseteq_A$

```
μX •
    if frameStack = ∅ ⟶ Skip
    ▯ frameStack ≠ ∅ ⟶
        if ···
        ▯ pc = i ⟶
            μY • A ; pc := i ; Poll ; Y
        ...
        fi ; Poll ; X
    fi
```

A.1.4 Resolve Method Calls

Rule [refine-invokespecial]**.**

$$\begin{array}{l}\{pc = i\};\\ \mathit{HandleInvokespecialEPC}(cpi)\end{array} \sqsubseteq_A \begin{array}{l}\{pc = i\}\ ;\ \mathbf{var}\ \mathit{poppedArgs} : \operatorname{seq} \mathit{Word} \bullet\\ \quad \big(\exists\, \mathit{argsToPop?} == \mathit{methodArguments}\ m + 1 \bullet\\ \qquad \mathit{InterpreterStackFrameInvoke}\big);\\ \quad \mathit{Invoke}(c, m, \mathit{poppedArgs})\end{array}$$

where $m : \mathit{MethodID}$ and $c : \mathit{ClassID}$ are such that

$$\begin{array}{l}\exists\, c_0 : \mathit{Class};\ m_0 : \mathit{MethodID} \mid c_0 \in \operatorname{ran} cs \land m_0 \in \operatorname{dom} c_0.\mathit{methodEntry} \bullet\\ \quad cpi \in \mathit{methodRefIndices}\ c_0 \land\\ \quad (\exists\, c_1 : \mathit{ClassID} \mid c_0.\mathit{constantPool}\ cpi = \mathit{MethodRef}\ (c_1, m) \bullet\\ \qquad (((\mathit{thisClassID}\ c_0, c_1) \in \mathit{subclassRel}\ cs\\ \qquad \land\ c_1 \neq \mathit{thisClassID}\ c_0\\ \qquad \land\ m \notin \mathit{initialisationMethodIDs})\\ \qquad\quad \Rightarrow c = \mathit{superClassID}\ c_0) \land\\ \qquad (((\mathit{thisClassID}\ c_0, c_1) \notin \mathit{subclassRel}\ cs\\ \qquad \lor\ c_1 = \mathit{thisClassID}\ c_0\\ \qquad \lor\ m \in \mathit{initialisationMethodIDs})\\ \qquad\quad \Rightarrow c = c_1)) \land\\ \quad i \in c_0.\mathit{methodEntry}\ m_0 \mathrel{..} c_0.\mathit{methodEnd}\ m_0.\end{array}$$

Rule [refine-invokestatic]**.**

$$\begin{array}{l}\{pc = i\};\\ \mathit{HandleInvokestaticEPC}(cpi)\end{array} \sqsubseteq_A \begin{array}{l}\{pc = i\}\ ;\ \mathbf{var}\ \mathit{poppedArgs} : \operatorname{seq} \mathit{Word} \bullet\\ \quad \big(\exists\, \mathit{argsToPop?} == \mathit{methodArguments}\ m \bullet\\ \qquad \mathit{InterpreterStackFrameInvoke}\big);\\ \quad \mathit{Invoke}(c, m, \mathit{poppedArgs})\end{array}$$

where $m : \mathit{MethodID}$ and $c : \mathit{ClassID}$ are such that

$$\begin{array}{l}\exists\, c_0 : \mathit{Class} \mid c_0 \in \operatorname{ran} cs \bullet\\ \quad (\exists\, m_0 : \mathit{MethodID} \mid m_0 \in \operatorname{dom} c_0.\mathit{methodEntry} \bullet\\ \qquad i \in c_0.\mathit{methodEntry}\ m_0 \mathrel{..} c_0.\mathit{methodEnd}\ m_0)\\ \quad cpi \in \mathit{methodRefIndices}\ c_0 \land c_0.\mathit{constantPool}\ cpi = \mathit{MethodRef}\ (c, m).\end{array}$$

Rule [refine-invokevirtual]**.** Given $i : \mathit{ProgramAddress}$,

$$
\begin{array}{l}
\{pc = j\};\\
\mathit{HandleInvokevirtualEPC}(cpi)
\end{array}
\sqsubseteq_A
\begin{array}{l}
\mathbf{var}\ \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet\\
\big(\exists\ \mathit{argsToPop?} == \mathit{methodArguments}\ m \bullet\\
\quad \mathit{InterpreterStackFrameInvoke}\big);\\
\mathit{getClassIDOf}!(\mathit{head}\ \mathit{poppedArgs})?\mathit{cid} \longrightarrow\\
\mathbf{if}\ \mathit{cid} = c_1 \longrightarrow\\
\quad \{(\mathit{last}\ \mathit{frameStack}).\mathit{storedPC} = j + 1\};\\
\quad \mathit{Invoke}(c_1, m, \mathit{poppedArgs})\\
\ldots\\
\ [\!]\ \mathit{cid} = c_n \longrightarrow\\
\quad \{(\mathit{last}\ \mathit{frameStack}).\mathit{storedPC} = j + 1\};\\
\quad \mathit{Invoke}(c_n, m, \mathit{poppedArgs})\\
\mathbf{fi}
\end{array}
$$

where $m : \mathit{MethodID}$ and $c_1, \ldots, c_n : \mathit{ClassID}$ are such that

$$
\begin{array}{l}
\exists\ c_0 : \mathit{Class};\ m_0 : \mathit{MethodID} \mid c_0 \in \mathrm{ran}\ cs \land m_0 \in \mathrm{dom}\ c_0.\mathit{methodEntry} \bullet\\
\quad cpi \in \mathit{methodRefIndices}\ c_0 \land\\
\quad j \in c_0.\mathit{methodEntry}\ m_0 \mathrel{..} c_0.\mathit{methodEnd}\ m_0 \land\\
\quad \exists\ c : \mathit{ClassID} \bullet c_0.\mathit{constantPool}\ cpi = \mathit{MethodRef}\ (c, m) \land\\
\quad \{x : \mathit{ClassID} \mid (x, c) \in \mathit{subclassRel}\ cs \land x \in \mathit{instCS}\} = \{c_1, \ldots, c_n\}
\end{array}
$$

Rule [resolve-special-method]**.** If c, m match one of the rows of Table 5.2, then

$$
\begin{array}{l}
\{pc = i\}\ ;\ (\mathbf{var}\ \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet\\
\big(\exists\ \mathit{argsToPop?} == e \bullet\\
\quad \mathit{InterpreterStackFrameInvoke}\big);\\
\mathit{Invoke}(c, m, \mathit{poppedArgs}))
\end{array}
\sqsubseteq_A
\begin{array}{l}
(\mathbf{var}\ \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet\\
\big(\exists\ \mathit{argsToPop?} == e \bullet\\
\quad \mathit{InterpreterStackFrameInvoke}\big);\\
\mathit{specialMethodAction}(c, m));\\
pc := i + 1
\end{array}
$$

where *specialMethodAction* is the syntactic function defined by Table 5.2.

Rule [resolve-normal-method]**.** Given $i : \mathit{ProgramAddress}$, if

- $$\{\mathit{frameStack} \neq \varnothing\} \; ; \; A \\ = \\ \{\mathit{frameStack} \neq \varnothing\} \; ; \; A \; ; \; \{\mathit{frameStack} \neq \varnothing\},$$

- $\mathit{methodID} = m \land \mathit{classID} = c \Rightarrow \mathbf{pre}\, \mathit{ResolveMethod}$ and there is $\mathit{classInfo} : \mathit{Class}$ such that

  $$\{\mathit{methodID} = m \land \mathit{classID} = c\} \; ; \; \big(\mathit{ResolveMethod}\big) \\ = \\ \{\mathit{methodID} = m \land \mathit{classID} = c\} \; ; \; \big(\mathit{ResolveMethod}\big); \\ \{\mathit{class} = \mathit{classInfo} \land \mathit{class}.\mathit{methodEntry}\, m = k\},$$

- for any $x : \mathit{ProgramAddress}$,

  $$\{(\mathit{last}\, (\mathit{front}\, \mathit{frameStack})).\mathit{storedPC} = x\} \; ; \; M \\ = \\ \{(\mathit{last}\, (\mathit{front}\, \mathit{frameStack})).\mathit{storedPC} = x\} \; ; \; M \; ; \; \{\mathit{pc} = x\},$$

- m and c do not match any of the conditions in Table 5.2,

then,

```
$\mu X \bullet$
  $\mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip}$
  $[\!]\, \mathit{frameStack} \neq \varnothing \longrightarrow$
    $\mathbf{if} \cdots$
    $[\!]\, \mathit{pc} = i \longrightarrow A \; ; \; \{\mathit{pc} = j\};$
      $\mathbf{var}\, \mathit{poppedArgs} : \mathrm{seq}\, \mathit{Word} \bullet$
      $\big(\exists\, \mathit{argsToPop}? == e \bullet$
         $\mathit{InterpreterStackFrameInvoke}\big);$
      $\mathit{Invoke}(c, m, \mathit{poppedArgs})$
    $[\!]\, \mathit{pc} = k \longrightarrow M$
    $\cdots$
    $\mathbf{fi} ; \mathit{Poll} \; ; \; X$
  $\mathbf{fi}$
```

$\sqsubseteq_A$

```
$\mu X \bullet$
  $\mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip}$
  $[\!]\, \mathit{frameStack} \neq \varnothing \longrightarrow$
    $\mathbf{if} \cdots$
    $[\!]\, \mathit{pc} = i \longrightarrow A;$
      $(\mathbf{var}\, \mathit{poppedArgs} : \mathrm{seq}\, \mathit{Word} \bullet$
      $\big(\exists\, \mathit{argsToPop}? == e \bullet$
         $\mathit{InterpreterStackFrameInvoke}\big);$
      $\mathit{CheckSynchronizedInvoke}($
         $\mathit{classInfo}, m, \mathit{poppedArgs});$
      $\big(\mathit{InterpreterNewStackFrame}[$
         $\mathit{classInfo} / \mathit{class}?,$
         $m / \mathit{methodID}?,$
         $\mathit{poppedArgs} / \mathit{methodArgs}?]\big);$
      $\mathit{Poll} \; ; \; M) \; ; \; \mathit{pc} := j + 1$
    $[\!]\, \mathit{pc} = k \longrightarrow M$
    $\cdots$
    $\mathbf{fi} ; \mathit{Poll} \; ; \; X$
  $\mathbf{fi}$
```

Rule [*CheckSynchronizedInvoke*-sync-refinement]**.**
If $m \in c.synchronizedMethods \land m \notin c.staticMethods$, then

$$CheckSynchronizedInvoke(c, m, args) \ \sqsubseteq_A \ \begin{array}{l} takeLock!(head\ args) \\ \longrightarrow takeLockRet \longrightarrow \mathbf{Skip} \end{array}$$

Rule [*CheckSynchronizedInvoke*-nonsync-refinement]**.**
If $m \notin c.synchronizedMethods \lor m \in c.staticMethods$, then

$$CheckSynchronizedInvoke(c, m, args) \ \sqsubseteq_A \ \mathbf{Skip}$$

Rule [resolve-normal-method-branch]**.** Given $i : ProgramAddress$ and $c_\ell : ClassID$, if

- $$\{frameStack \neq \varnothing\}\ ;\ A \\ = \\ \{frameStack \neq \varnothing\}\ ;\ A\ ;\ \{frameStack \neq \varnothing\},$$

- $methodID = m \land classID = c_\ell \Rightarrow \mathbf{pre}\, ResolveMethod$ and there is $classInfo : Class$ such that

$$\{methodID = m \land classID = c_\ell\}\ ;\ \big(ResolveMethod\big) \\ = \\ \{methodID = m \land classID = c_\ell\}\ ;\ \big(ResolveMethod\big); \\ \{class = classInfo \land class.methodEntry\ m = k\},$$

- for any $x : ProgramAddress$,

$$\{(last\,(front\,frameStack)).storedPC = x\}\ ;\ M \\ = \\ \{(last\,(front\,frameStack)).storedPC = x\}\ ;\ M\ ;\ \{pc = x\},$$

- m and c do not match any of the conditions in Table 5.2,

then,

```
μX •
  if frameStack = ∅ ⟶ Skip
  ▯ frameStack ≠ ∅ ⟶
    if ···
    ▯ pc = i ⟶ A;
      var poppedArgs : seq Word •
      (∃ argsToPop? == e •
        InterpreterStackFrameInvoke);
      if cid = c_1 ⟶ A_1
      ···
      ▯ cid = c_ℓ ⟶
        {(last frameStack).storedPC = j + 1};
        Invoke(c_ℓ, m, poppedArgs, False)
      ···
      ▯ cid = c_n ⟶ A_n
      fi
    ▯ pc = k ⟶ M
    ···
    fi ; Poll ; X
  fi
```

$\sqsubseteq_A$

```
μX •
  if frameStack = ∅ ⟶ Skip
  ▯ frameStack ≠ ∅ ⟶
    if ···
    ▯ pc = i ⟶ A ; {pc = j};
      var poppedArgs : seq Word •
      (∃ argsToPop? == e •
        InterpreterStackFrameInvoke);
      if cid = c_1 ⟶ A_1
      ···
      ▯ cid = c_ℓ ⟶
        CheckSynchronizedInvoke(
          classInfo, m, poppedArgs);
        (InterpreterNewStackFrame[
          classInfo/class?,
          m/methodID?,
          poppedArgs/methodArgs?]);
        Poll ; M ; pc := j + 1
      ···
      ▯ cid = c_n ⟶ A_n
      fi)
    ▯ pc = k ⟶ M
    ···
    fi ; Poll ; X
  fi
```

Rule [virtual-method-call-dist].

$$\begin{array}{l}
(\mathbf{var}\, \mathit{poppedArgs} : \mathrm{seq}\, \mathit{Word} \bullet P \\
\mathit{getClassIDOf}!(\mathit{head}\, \mathit{poppedArgs})?\mathit{cid} \longrightarrow \\
\mathbf{if}\ \mathit{cid} = c_1 \longrightarrow A_1 \mathbin{;}\ \mathit{pc} := x \\
\ldots \\
\ \square\ \mathit{cid} = c_n \longrightarrow A_\ell \mathbin{;}\ \mathit{pc} := x \\
\mathbf{fi})
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
(\mathbf{var}\, \mathit{poppedArgs} : \mathrm{seq}\, \mathit{Word} \bullet P \\
\mathit{getClassIDOf}!(\mathit{head}\, \mathit{poppedArgs})?\mathit{cid} \longrightarrow \\
\mathbf{if}\ \mathit{cid} = c_1 \longrightarrow A_1 \\
\ldots \\
\ \square\ \mathit{cid} = c_n \longrightarrow A_\ell \\
\mathbf{fi}) \mathbin{;}\ \mathit{pc} := x
\end{array}$$

A.1.5 Refine Main Actions

Rule [*StartInterpreter-Running*-refinement]. If $(c_1, m_1), \ldots, (c_n, m_n)$ are the only $\mathit{ClassID} \times \mathit{MethodID}$ values such that $\mathit{classID} = c_i \land \mathit{methodID} = m_i \Rightarrow \mathbf{pre}\, \mathit{ResolveMethod}$, and for each $i \in \{1 \mathinner{..} n\}$, there exists $\mathit{classInfo}_i : \mathit{Class}$ and $\mathit{entry}_i : \mathit{ProgramAddress}$ such that,

$$\begin{array}{l}
\{\mathit{classID} = c_i \land \mathit{methodID} = m_i\} \mathbin{;}\ \mathit{ResolveMethod} \\
= \\
\{\mathit{classID} = c_i \land \mathit{methodID} = m_i\} \mathbin{;}\ \mathit{ResolveMethod}; \\
\qquad \{\mathit{class} = \mathit{classInfo}_i \land \mathit{classInfo}_i.\mathit{methodEntry}\, m_i = \mathit{entry}_i\},
\end{array}$$

and, for each $i \in \{1 \mathinner{..} n\}$,

$$\begin{array}{l}
\{\#\, \mathit{frameStack} = 1\} \mathbin{;}\ M_i \\
= \\
\{\#\, \mathit{frameStack} = 1\} \mathbin{;}\ M_i \mathbin{;}\ \{\mathit{framestack} = \varnothing\},
\end{array}$$

then,

$$\begin{array}{l}
\{\mathit{frameStack} = \varnothing\}; \\
\mathit{StartInterpreter} \mathbin{;}\ \mathit{Poll} \mathbin{;}\ \mu X \bullet \\
\quad \mathbf{if}\, \mathit{frameStack} = \varnothing \longrightarrow \mathbf{Skip} \\
\quad \square\ \mathit{framestack} \neq \varnothing \longrightarrow \\
\qquad \mathbf{if}\, \mathit{pc} = \mathit{entry}_1 \longrightarrow M_1 \\
\qquad \ldots \\
\qquad \square\ \mathit{pc} = \mathit{entry}_n \longrightarrow M_n \\
\qquad \mathbf{fi} \mathbin{;} \mathit{Poll} \mathbin{;}\ X \\
\quad \mathbf{fi}
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
\mathit{executeMethod}?t : (t = \mathit{thread})?c?m?a \longrightarrow \\
(\mathbf{val}\, \mathit{classID} : \mathit{ClassID}; \\
\mathbf{val}\, \mathit{methodID} : \mathit{MethodID}; \\
\mathbf{val}\, \mathit{methodArgs} : \mathrm{seq}\, \mathit{Word} \bullet \\
\mathbf{if}\ (\mathit{classID}, \mathit{methodID}) = (c_1, m_1) \longrightarrow \\
\quad \mathit{InterpreterNewStackFrame}[\\
\qquad \mathit{classInfo}_1 / \mathit{class}?, \\
\qquad m_1 / \mathit{methodID}?] \mathbin{;}\ \mathit{Poll} \mathbin{;}\ M_1 \\
\ldots \\
\square\ (\mathit{classID}, \mathit{methodID}) = (c_n, m_n) \longrightarrow \\
\quad \mathit{InterpreterNewStackFrame}[\\
\qquad \mathit{classInfo}_n / \mathit{class}?, \\
\qquad m_n / \mathit{methodID}?] \mathbin{;}\ \mathit{Poll} \mathbin{;}\ M_n \\
\mathbf{fi})(c, m, a) \mathbin{;}\ \mathit{Poll}
\end{array}$$

A.2 Elimination of Frame Stack

A.2.1 Remove Launcher Returns

Rule [conditional-dist]**.** Given an action X,

$$\begin{array}{lcl} \mathbf{var}\ value1, value2 : Word \bullet A; & & (\mathbf{var}\ value1, value2 : Word \bullet A; \\ \quad \mathbf{if}\ b \longrightarrow B\ ;\ X & & \quad \mathbf{if}\ b \longrightarrow B \\ \quad [\!]\ c \longrightarrow C\ ;\ X & \sqsubseteq_A & \quad [\!]\ c \longrightarrow C \\ \quad \mathbf{fi} & & \quad \mathbf{fi})\ ;\ X \end{array}$$

Algorithm 15 RedefineMethodExcludingReturn(*methodName*,*returnAction*)

```
1: $methodBody \leftarrow$ ActionBody($methodName$)
2: match $methodBody$ with ($A$ ; $returnAction$) then
3:     apply Law [action-intro]($methodName'$, $A$)
4: apply Law [copy-rule]($methodName'$) in reverse to $methodBody$
5: exhaustively apply Law [copy-rule]($methodName$)
6: apply Law [action-intro]($methodName$, $methodBody$) in reverse
7: apply Law [action-rename]($methodName$, $methodName'$)
```

Algorithm 16 IntroduceFrameStackAssumptions

```
1: apply Rule [$InterpreterInitEPC$-$frameStack$-assump-intro]
2: exhaustively apply
3:         Rule [$frameStack$-assump-non-return-dist]
4:         Rule [$frameStack$-assump-return-dist-rule]
5:         Rule [$frameStack$-assump-NewStackFrame-dist]
6:         Rule [restricted-assump-alt-distl]
7:         Rule [restricted-assump-alt-distr]
8:         Rule [restricted-assump-var-distl]
9:         Rule [restricted-assump-var-distr]
10:        Rule [restricted-assump-output-prefix-distl]
11:        Rule [restricted-assump-output-prefix-distr]
12:        Rule [restricted-assump-input-prefix-distl]
13:        Rule [restricted-assump-input-prefix-distr]
14:        Rule [restricted-assump-infinite-loop-distl]
15:        Rule [restricted-assump-infinite-loop-distr]
16:        Rule [restricted-assump-while-loop-distl]
17:        Rule [restricted-assump-while-loop-distr]
18:        Rule [restricted-assump-do-while-loop-distl]
19:        Rule [restricted-assump-do-while-loop-distr]
20:        Rule [restricted-assump-mid-while-loop-distl]
21:        Rule [restricted-assump-mid-while-loop-distr]
22:        Rule [restricted-assump-extchoice-distl]
23:        Rule [restricted-assump-extchoice-distr]
24:        Rule [restricted-assump-guard-dist]
25:        Rule [restricted-assump-assign-dist]
```

Rule [*InterpreterInitEPC*-*frameStack*-assump-intro]**.**

$$\big(\mathit{InterpreterInitEPC}\big) \sqsubseteq_A \big(\mathit{InterpreterInitEPC}\big)\ ;\ \{\#\mathit{frameStack} = 0\}$$

Rule [*frameStack*-assump-NewStackFrame-dist]**.**

$$\begin{array}{l}\{\#\mathit{frameStack} = k\};\\ \big(\mathit{InterpreterNewStackFrame}[\\ \quad c/\mathit{class}?,\quad m/\mathit{methodID}?,\\ \quad \mathit{args}/\mathit{methodArgs}?]\big)\end{array} \sqsubseteq_A \begin{array}{l}\{\#\mathit{frameStack} = k\};\\ \big(\mathit{InterpreterNewStackFrame}[\\ \quad c/\mathit{class}?,\quad m/\mathit{methodID}?,\\ \quad \mathit{args}/\mathit{methodArgs}?]\big);\\ \{\#\mathit{frameStack} = k+1\}\end{array}$$

Rule [*frameStack*-assump-non-return-dist]**.** If A is one of

- **Skip**
- *Poll*,
- *HandleAconst_nullEPC*,
- *HandleDupEPC*,
- $\mathit{HandleAloadEPC}(\mathit{lvi})$,
- $\mathit{HandleAstoreEPC}(\mathit{lvi})$,
- *HandleIaddEPC*,
- $\mathit{HandleIconstEPC}(n)$,
- *HandleInegEPC*,
- $\big(\mathit{InterpreterPopEPC}\big)$,
- $\big(\mathit{InterpreterPushEPC}\big)$,
- $\big(\exists\, \mathit{argsToPop}? == m \bullet \mathit{InterpreterStackFrameInvoke}\big)$,
- $\mathit{HandleNewEPC}(\mathit{cpi})$,
- $\mathit{HandleGetfieldEPC}(\mathit{cpi})$,
- $\mathit{HandlePutfieldEPC}(\mathit{cpi})$,
- $\mathit{HandleGetstaticEPC}(\mathit{cpi})$, or
- $\mathit{HandlePutstaticEPC}(\mathit{cpi})$,

and B does not begin with $\{\#\mathit{frameStack} = k\}$, then

$$\{\#\mathit{frameStack} = k\}\ ;\ A\ ;\ B \sqsubseteq_A \{\#\mathit{frameStack} = k\}\ ;\ A\ ;\ \{\#\mathit{frameStack} = k\}\ ;\ B$$

Rule [*frameStack*-assump-return-dist-rule]**.** If A is *HandleAreturnEPC* or *HandleReturnEPC*, B does not begin with $\{\#\mathit{frameStack} = k\}$, and $k > 0$ then

$$\{\#\mathit{frameStack} = k\}\ ;\ A\ ;\ B \sqsubseteq_A \begin{array}{l}\{\#\mathit{frameStack} = k\}\ ;\ A;\\ \quad \{\#\mathit{frameStack} = k-1\}\ ;\ B\end{array}$$

Rule [restricted-assump-alt-distl]**.** If no A_i begins with $\{h\}$ then

$$\{h\}\ ;\ \mathbf{if}\ \square_i\ g_i \longrightarrow A_i\ \mathbf{fi} = \{h\}\ ;\ \mathbf{if}\ \square_i\ g_i \longrightarrow \{h\}\ ;\ A_i\ \mathbf{fi}$$

Rule [restricted-assump-alt-distr]**.** If no A_i begins with $\{h\}$ then

$$\mathbf{if}\ \square_i\ g_i \longrightarrow A_i\ ;\ \{h\}\ \mathbf{fi} = \{h\}\ ;\ \mathbf{if}\ \square_i\ g_i \longrightarrow A_i\ \mathbf{fi}\ ;\ \{h\}$$

Rule [restricted-assump-var-distl]**.** If A does not begin with $\{h\}$ then

$$\{h\}\ ;\ (\mathbf{var}\ x : T \bullet A) = \{h\}\ ;\ (\mathbf{var}\ x : T \bullet \{h\}\ ;\ A)$$

Rule [restricted-assump-var-distr]. If B does not begin with $\{h\}$ then

$$(\mathbf{var}\, x : T \bullet A \, ; \; \{h\}) \, ; \; B = (\mathbf{var}\, x : T \bullet A \, ; \; \{h\}) \, ; \; \{h\} \, ; \; B$$

Rule [restricted-assump-output-prefix-distl]. If A does not begin with $\{g\}$ then

$$\{g\} \, ; \; c!x \longrightarrow A = \{g\} \, ; \; c!x \longrightarrow \{g\} \, ; \; A$$

Rule [restricted-assump-output-prefix-distr]. If B does not begin with $\{g\}$ then

$$(c!x \longrightarrow A \, ; \; \{g\}) \, ; \; B = (c!x \longrightarrow A \, ; \; \{g\}) \, ; \; \{g\} \, ; \; B$$

Rule [restricted-assump-input-prefix-distl]. If A does not begin with $\{g\}$ and x is not free in $\{g\}$ then

$$\{g\} \, ; \; c?x \longrightarrow A = \{g\} \, ; \; c?x \longrightarrow \{g\} \, ; \; A$$

Rule [restricted-assump-input-prefix-distr]. If B does not begin with $\{g\}$ and x is not free in $\{g\}$ then

$$(c?x \longrightarrow A \, ; \; \{g\}) \, ; \; B = (c?x \longrightarrow A \, ; \; \{g\}) \, ; \; \{g\} \, ; \; B$$

Rule [restricted-assump-infinite-loop-distl]. If A does not begin with $\{g\}$ and $\{g\}$; $A \sqsubseteq_A A$; $\{g\}$ then

$$\{g\} \, ; \; (\mu X \bullet A \, ; \; X) \sqsubseteq_A \{g\} \, ; \; (\mu X \bullet \{g\} \, ; \; A \, ; \; X)$$

Rule [restricted-assump-infinite-loop-distr]. If B does not begin with $\{g\}$ then

$$(\mu X \bullet A \, ; \; \{g\} \, ; \; X) \, ; \; B = (\mu X \bullet A \, ; \; \{g\} \, ; \; X) \, ; \; \{g\} \, ; \; B$$

Rule [restricted-assump-mid-while-loop-distl]. If A does not begin with $\{g\}$, $\{g\}$; $A \sqsubseteq_A A$; $\{g\}$, $\{g\} \, ; \; B \sqsubseteq_A B \, ; \; \{g\}$, then

$$\begin{array}{l} \{g\} \, ; \; (\mu X \bullet A; \\ \qquad \mathbf{if}\ h \longrightarrow B \, ; \; X \\ \qquad [\!]\ \neg\, h \longrightarrow \mathbf{Skip} \\ \qquad \mathbf{fi}) \end{array} \sqsubseteq_A \begin{array}{l} \{g\} \, ; \; (\mu X \bullet \{g\} \, ; \; A; \\ \qquad \mathbf{if}\ h \longrightarrow B \, ; \; X \\ \qquad [\!]\ \neg\, h \longrightarrow \mathbf{Skip} \\ \qquad \mathbf{fi}) \end{array}$$

Rule [restricted-assump-mid-while-loop-distr]. If C does not begin with $\{g\}$, $\{g\}$; $A \sqsubseteq_A A$; $\{g\}$, $\{g\} \, ; \; B \sqsubseteq_A B \, ; \; \{g\}$, then

$$\begin{array}{l} (\mu X \bullet A; \\ \qquad \mathbf{if}\ h \longrightarrow B \, ; \; \{g\} \, ; \; X \\ \qquad [\!]\ \neg\, h \longrightarrow \mathbf{Skip} \, ; \; \{g\} \\ \qquad \mathbf{fi}) \, ; \; C \end{array} \sqsubseteq_A \begin{array}{l} (\mu X \bullet A; \\ \qquad \mathbf{if}\ h \longrightarrow B \, ; \; \{g\} \, ; \; X \\ \qquad [\!]\ \neg\, h \longrightarrow \mathbf{Skip} \, ; \; \{g\} \\ \qquad \mathbf{fi}) \, ; \; \{g\} \, ; \; C \end{array}$$

Rule [restricted-assump-do-while-loop-distl]. If A does not begin with $\{g\}$ and $\{g\} \, ; \; A \sqsubseteq_A A \, ; \; \{g\}$, then

$$\begin{array}{l} \{g\} \, ; \; (\mu X \bullet A; \\ \qquad \mathbf{if}\ h \longrightarrow X \\ \qquad [\!]\ \neg\, h \longrightarrow \mathbf{Skip} \\ \qquad \mathbf{fi}) \end{array} \sqsubseteq_A \begin{array}{l} \{g\} \, ; \; (\mu X \bullet \{g\} \, ; \; A; \\ \qquad \mathbf{if}\ h \longrightarrow X \\ \qquad [\!]\ \neg\, h \longrightarrow \mathbf{Skip} \\ \qquad \mathbf{fi}) \end{array}$$

Rule [restricted-assump-do-while-loop-distr]. If B does not begin with $\{g\}$ and $\{g\}\ ;\ A \sqsubseteq_A A\ ;\ \{g\}$, then

$$\begin{array}{l}(\mu X \bullet A;\\ \quad \mathbf{if}\ h \longrightarrow \{g\}\ ;\ X\\ \quad [\!]\ \neg\ h\ ;\ \mathbf{Skip}\ ;\ \{g\}\\ \quad \mathbf{fi})\ ;\ B\end{array} \sqsubseteq_A \begin{array}{l}(\mu X \bullet A;\\ \quad \mathbf{if}\ h \longrightarrow \{g\}\ ;\ X\\ \quad [\!]\ \neg\ h\ ;\ \mathbf{Skip}\ ;\ \{g\}\\ \quad \mathbf{fi})\ ;\ \{g\}\ ;\ B\end{array}$$

Rule [restricted-assump-while-loop-distl]. If A does not begin with $\{g\}$ and $\{g\};\ A \sqsubseteq_A A;\ \{g\}$, then

$$\begin{array}{l}\{g\}\ ;\ (\mu X \bullet\\ \quad \mathbf{if}\ h \longrightarrow A\ ;\ X\\ \quad [\!]\ \neg\ h \longrightarrow \mathbf{Skip}\\ \quad \mathbf{fi})\end{array} \sqsubseteq_A \begin{array}{l}\{g\}\ ;\ (\mu X \bullet \{g\};\\ \quad \mathbf{if}\ h \longrightarrow A\ ;\ X\\ \quad [\!]\ \neg\ h \longrightarrow \mathbf{Skip}\\ \quad \mathbf{fi})\end{array}$$

Rule [restricted-assump-while-loop-distr]. If B does not begin with $\{g\}$ and $\{g\};\ A \sqsubseteq_A A;\ \{g\}$, then

$$\begin{array}{l}(\mu X \bullet\\ \quad \mathbf{if}\ h \longrightarrow A\ ;\ \{g\}\ ;\ X\\ \quad [\!]\ \neg\ h \longrightarrow \mathbf{Skip}\ ;\ \{g\}\\ \quad \mathbf{fi})\ ;\ B\end{array} \sqsubseteq_A \begin{array}{l}(\mu X \bullet\\ \quad \mathbf{if}\ h \longrightarrow A\ ;\ \{g\}\ ;\ X\\ \quad [\!]\ \neg\ h \longrightarrow \mathbf{Skip}\ ;\ \{g\}\\ \quad \mathbf{fi})\ ;\ \{g\}\ ;\ B\end{array}$$

Rule [restricted-assump-extchoice-distl]. If A and B do not begin with $\{g\}$ then

$$\{g\}\ ;\ (A \mathbin{\Box} B) \ \sqsubseteq_A\ \{g\}\ ;\ ((\{g\}\ ;\ A) \mathbin{\Box} (\{g\}\ ;\ B))$$

Rule [restricted-assump-extchoice-distr]. If C does not begin with $\{g\}$ then

$$((A\ ;\ \{g\}) \mathbin{\Box} (B\ ;\ \{g\}))\ ;\ C \ \sqsubseteq_A\ ((A\ ;\ \{g\}) \mathbin{\Box} (B\ ;\ \{g\}))\ ;\ \{g\}\ ;\ C$$

Rule [restricted-assump-guard-dist]. If A does not begin with $\{g\}$ then

$$\{g\}\ ;\ (h) \mathbin{\&} A \ =\ \{g\}\ ;\ (h) \mathbin{\&} \{g\}\ ;\ A$$

Rule [restricted-assump-assign-dist]. If B does not begin with $\{g\}$ and x is not free in g then

$$\{g\}\ ;\ x := e\ ;\ B \ =\ \{g\}\ ;\ x := e\ ;\ \{g\}\ ;\ B$$

Rule [refine-*HandleReturnEPC*-empty-*frameStack*].

$$\begin{array}{l}\{\#\mathit{frameStack} = 1\};\\ \mathit{HandleReturnEPC}\end{array} \sqsubseteq_A \begin{array}{l}\mathbf{var}\ \mathit{returnValue} : \mathit{Word} \bullet\\ \big(\mathit{InterpreterReturnEPC}\big);\\ \mathit{executeMethodRet}!\mathit{thread}!\mathit{returnValue} \longrightarrow \mathbf{Skip}\end{array}$$

Rule [refine-*HandleReturnEPC*-nonempty-*frameStack*]. If $k > 1$ then

$$\begin{array}{l}\{\#\mathit{frameStack} = k\};\\ \mathit{HandleReturnEPC}\end{array} \sqsubseteq_A \big(\mathit{InterpreterReturnEPC}\big)$$

Rule [refine-*HandleAreturnEPC*-empty-*frameStack*].

$$\begin{array}{l}\{\#\mathit{frameStack} = 1\};\\ \mathit{HandleAreturnEPC}\end{array} \sqsubseteq_A \begin{array}{l}\mathbf{var}\ \mathit{returnValue} : \mathit{Word} \bullet\\ \big(\mathit{InterpreterAreturn2EPC}\big);\\ \mathit{executeMethodRet}!\mathit{thread}!\mathit{returnValue} \longrightarrow \mathbf{Skip}\end{array}$$

Rule [refine-*HandleAreturnEPC*-nonempty-*frameStack*]**.** If $k > 1$ then

$$\begin{array}{l}\{\# \mathit{frameStack} = k\};\\ \mathit{HandleAreturnEPC}\end{array} \sqsubseteq_A \left(\mathit{InterpreterAreturn1EPC}\right)$$

A.2.2 Localise Stack Frames

Rule [*InterpreterReturn*-args-intro]. Given an action name M and $n : \mathbb{N}$, if $\mathit{arg}1, \ldots, \mathit{arg}{<}n{>}$ are not free in M, are distinct from c, m and args, and $\# \mathit{args} = n$, then

$$\begin{array}{l}\big(\mathit{InterpreterNewStackFrame}[\\ \qquad c/\mathit{class}?,\\ \qquad m/\mathit{methodID}?,\\ \qquad \mathit{args}/\mathit{methodArgs}?]\big);\\ \mathit{Poll} \;;\; M \;;\; \big(\mathit{InterpreterReturn}\big)\end{array} \sqsubseteq_A \begin{array}{l}(\mathbf{val}\ \mathit{arg}1, \ldots, \mathit{arg}{<}n{>} : \mathit{Word} \bullet\\ \qquad \big(\exists\, \mathit{methodArgs}? == \langle \mathit{arg}1, \ldots, \mathit{arg}{<}n{>}\rangle \bullet\\ \qquad\qquad \mathit{InterpreterNewStackFrame}[\\ \qquad\qquad\qquad c/\mathit{class}?,\\ \qquad\qquad\qquad m/\mathit{methodID}?]\big);\\ \qquad \mathit{Poll} \;;\; M \;;\; \big(\mathit{InterpreterReturn}\big)\\ )(\mathit{args}\,1, \ldots, \mathit{args}\,n)\end{array}$$

Algorithm 17 RedefineMethodToIncludeParameters(*nethodName*)

```
1: match  $(\mathbf{val}\ \mathit{arg}1, \ldots, \mathit{arg}{<}n{>} : \mathit{Word} \bullet$                then
              $\big(\exists\, \mathit{methodArgs}? == \langle \mathit{arg}1, \ldots, \mathit{arg}{<}n{>}\rangle \bullet$
                  $\mathit{InterpreterNewStackFrame}[c/\mathit{class}?, m/\mathit{methodID}?]\big);$
              $\mathit{Poll} \;;\; \mathit{methodName} \;;\; \big(\mathit{InterpreterReturn}\big))(\mathit{args}\,1, \ldots, \mathit{args}\,n)$
2:     apply Law [action-intro]($\mathit{methodName}'$,  $(\mathbf{val}\ \mathit{arg}1, \ldots, \mathit{arg}{<}n{>} : \mathit{Word} \bullet$        )
                                                  $\big(\exists\, \mathit{methodArgs}?$
                                                        $== \langle \mathit{arg}1, \ldots, \mathit{arg}{<}n{>}\rangle \bullet$
                                                      $\mathit{InterpreterNewStackFrame}[$
                                                        $c/\mathit{class}?, m/\mathit{methodID}?]\big);$
                                                  $\mathit{Poll} \;;\; \mathit{methodName};$
                                                  $\big(\mathit{InterpreterReturn}\big))$
3: exhaustively apply Law [copy-rule]($\mathit{methodName}'$) in reverse
4: apply Law [copy-rule]($\mathit{methodName}$) to ActionBody($\mathit{methodName}'$)
5: apply Law [action-intro]($\mathit{methodName}$, $\mathit{methodBody}$) in reverse
6: apply Law [action-rename]($\mathit{methodName}'$, $\mathit{methodName}$)
```

Rule [*InterpreterReturn*-*stackFrame*-intro]**.** Given $n : \mathbb{N}$, if the only occurrences of *frameStack* in A are in the expression *last frameStack*, the length of *frameStack* does not change throughout A, and *stackFrame* is not free in A, then

$$\begin{array}{l} \big(\exists\, methodArgs? == \langle arg1, \ldots, arg{<}n{>}\rangle \bullet \\ \quad InterpreterNewStackFrame[\\ \qquad c/class?, \\ \qquad m/methodID?]\big); \\ A\ ;\ \big(InterpreterReturn\big) \end{array} \sqsubseteq_A \begin{array}{l} \mathbf{var}\ stackFrame : StackFrameEPC \bullet \\ \quad \big(Init{<}c{>}\_{<}m{>}SF\big); \\ \quad A[stackFrame/last\ frameStack, \\ \qquad stackFrame'/last\ frameStack'] \end{array}$$

where *Init<c>_<m>SF* is defined by

$$\begin{array}{|l} \hline Init{<}c{>}\_{<}m{>}SF \\ arg1?, \ldots, arg{<}n{>}? : Word \\ stackFrame' : StackFrameEPC \\ \hline \langle arg1?, \ldots, arg{<}n{>}?\rangle \text{ prefix } stackFrame'.localVariables \\ \#\, stackFrame'.localVariables = c.methodLocals\ m \\ stackFrame'.operandStack = \langle\rangle \\ stackFrame'.frameClass = c \\ stackFrame'.stackSize = c.methodStackSize\ m \\ \hline \end{array}$$

A.2.3 Introduce Variables

Algorithm 18 IntroduceFrameClassAssumptions(A)

```
1: apply Rule [stackFrame-init-frameClass-assump-intro] to $A$
2: exhaustively apply to $A$
3:     Rule [frameClass-assump-dist]
4:     Rule [restricted-assump-alt-distl]
5:     Rule [restricted-assump-alt-distr]
6:     Rule [restricted-assump-var-distl]
7:     Rule [restricted-assump-var-distr]
8:     Rule [restricted-assump-output-prefix-distl]
9:     Rule [restricted-assump-output-prefix-distr]
10:    Rule [restricted-assump-input-prefix-distl]
11:    Rule [restricted-assump-input-prefix-distr]
12:    Rule [restricted-assump-infinite-loop-distl]
13:    Rule [restricted-assump-infinite-loop-distr]
14:    Rule [restricted-assump-while-loop-distl]
15:    Rule [restricted-assump-while-loop-distr]
16:    Rule [restricted-assump-do-while-loop-distl]
17:    Rule [restricted-assump-do-while-loop-distr]
18:    Rule [restricted-assump-mid-while-loop-distl]
19:    Rule [restricted-assump-mid-while-loop-distr]
```

Rule [*stackFrame*-init-*frameClass*-assump-intro]**.**

$$\begin{array}{l}\big([arg1?, \ldots, arg{<}n{>}? : Word; \\ \quad stackFrame' : StackFrameEPC \mid \\ \quad \langle arg1?, \ldots, arg{<}n{>}? \rangle \\ \qquad \subseteq stackFrame'.localVariables \land \\ \quad \# stackFrame'.localVariables = \ell \land \\ \quad stackFrame'.operandStack = \langle\rangle \land \\ \quad stackFrame'.frameClass = c \land \\ \quad stackFrame'.stackSize = s]\big)\end{array} \sqsubseteq_A \begin{array}{l}\big([arg1?, \ldots, arg{<}n{>}? : Word; \\ \quad stackFrame' : StackFrameEPC \mid \\ \quad \langle arg1?, \ldots, arg{<}n{>}? \rangle \\ \qquad \subseteq stackFrame'.localVariables \land \\ \quad \# stackFrame'.localVariables = \ell \land \\ \quad stackFrame'.operandStack = \langle\rangle \land \\ \quad stackFrame'.frameClass = c \land \\ \quad stackFrame'.stackSize = s]\big); \\ \{stackFrame.frameClass = c\}\end{array}$$

Rule [*frameClass*-assump-dist]**.** If A is one of

- **Skip**,
- $Poll$,
- $HandleAconst\_nullSF$,
- $HandleDupSF$,
- $HandleAloadSF(lvi)$,
- $HandleAstoreSF(lvi)$,
- $HandleIaddSF$,
- $HandleIconstSF(n)$,
- $HandleInegSF$,
- $\big(InterpreterPopSF\big)$,
- $\big(InterpreterPushSF\big)$,
- $\big(\exists\, argsToPop? == m \bullet InvokeSF\big)$

and B does not begin with $\{stackFrame.frameClass = c\}$, then

$$\{stackFrame.frameClass = c\} \;;\; A \;;\; B \;\sqsubseteq_A\; \begin{array}{l}\{stackFrame.frameClass = c\} \;;\; A; \\ \{stackFrame.frameClass = c\} \;;\; B\end{array}$$

Rule [refine-*PutfieldSF*]**.**

$$\begin{array}{l}\{stackFrame.frameClass = c\}; \\ PutfieldSF(cpi)\end{array} \sqsubseteq_A \begin{array}{l}(\mathbf{var}\ oid : ObjectID;\ value : Word \bullet \\ \quad \big(InterpreterPop[\\ \qquad stackFrame / last\ frameStack, \\ \qquad stackFrame' / last\ frameStack']\big); \\ \quad \big(InterpreterPop[\\ \qquad oid! / value!, \\ \qquad stackFrame / last\ frameStack, \\ \qquad stackFrame' / last\ frameStack']\big); \\ \quad putField!oid!cid!fid!value \longrightarrow \mathbf{Skip})\end{array}$$

where

$$\begin{array}{l}cpi \in fieldRefIndices\ c \land \\ c.constantPool\ cpi = FieldRef\ (cid, fid)\end{array}$$

Rule [refine-*GetfieldSF*].

$$\begin{array}{l}\{stackFrame.frameClass = c\};\\ GetfieldSF(cpi)\end{array} \quad \sqsubseteq_A \quad \begin{array}{l}(\mathbf{var}\ oid : ObjectID \bullet \\ \quad \big(InterpreterPop[\\ \qquad oid!/value!,\\ \qquad stackFrame/last\ frameStack,\\ \qquad stackFrame'/last\ frameStack']\big);\\ \quad getField!oid!cid!fid!\\ \quad \longrightarrow getFieldRet?value\\ \quad \longrightarrow \big(InterpreterPush[\\ \qquad stackFrame/last\ frameStack,\\ \qquad stackFrame'/last\ frameStack']\big))\end{array}$$

where

$$\begin{array}{l}cpi \in fieldRefIndices\ c\ \wedge\\ c.constantPool\ cpi = FieldRef\ (cid, fid)\end{array}$$

Rule [refine-*PutstaticSF*].

$$\begin{array}{l}\{stackFrame.frameClass = c\};\\ PutstaticSF(cpi)\end{array} \quad \sqsubseteq_A \quad \begin{array}{l}(\mathbf{var}\ value : Word \bullet \\ \quad \big(InterpreterPop[\\ \qquad stackFrame/last\ frameStack,\\ \qquad stackFrame'/last\ frameStack']\big);\\ \quad putStatic!cid!fid!value \longrightarrow \mathbf{Skip})\end{array}$$

where

$$\begin{array}{l}cpi \in fieldRefIndices\ c\ \wedge\\ c.constantPool\ cpi = FieldRef\ (cid, fid)\end{array}$$

Rule [refine-*GetstaticSF*].

$$\begin{array}{l}\{stackFrame.frameClass = c\};\\ GetstaticSF(cpi)\end{array} \quad \sqsubseteq_A \quad \begin{array}{l}getStatic!cid!fid\\ \longrightarrow getStaticRet?value\\ \longrightarrow \big(InterpreterPush[\\ \quad stackFrame/last\ frameStack,\\ \quad stackFrame'/last\ frameStack']\big))\end{array}$$

where

$$\begin{array}{l}cpi \in fieldRefIndices\ c\ \wedge\\ c.constantPool\ cpi = FieldRef\ (cid, fid)\end{array}$$

Rule [refine-*NewSF*]**.**

$$\begin{array}{l} \{\mathit{stackFrame.frameClass} = c\}; \\ \mathit{NewSF}(\mathit{cpi}) \end{array} \sqsubseteq_A \begin{array}{l} \mathit{newObject}!\mathit{thread}!\mathit{cid} \\ \longrightarrow \mathit{newObjectRet}?\mathit{oid} \\ \longrightarrow \big(\mathit{InterpreterPush}[\\ \quad \mathit{oid}/\mathit{value}?, \\ \quad \mathit{stackFrame}/\mathit{last}\,\mathit{frameStack}, \\ \quad \mathit{stackFrame}'/\mathit{last}\,\mathit{frameStack}']\big)\big) \end{array}$$

where

$$\begin{array}{l} \mathit{cpi} \in \mathit{ClassRefIndices}\ c \land \\ c.\mathit{constantPool}\ \mathit{cpi} = \mathit{ClassRef}\ \mathit{cid} \end{array}$$

Algorithm 19 IntroduceOperandStackAssumptions(A)

```
1: apply Rule [operandStack-init-frameClass-assump-intro] to $A$
2: exhaustively apply to $A$
3:     Rule [operandStack-assump-unchanged-dist]
4:     Rule [operandStack-assump-increment-dist]
5:     Rule [operandStack-assump-decrement-dist]
6:     Rule [operandStack-assump-InvokeSF-dist]
7:     Rule [restricted-assump-alt-distl]
8:     Rule [restricted-assump-alt-distr]
9:     Rule [restricted-assump-var-distl]
10:    Rule [restricted-assump-var-distr]
11:    Rule [restricted-assump-output-prefix-distl]
12:    Rule [restricted-assump-output-prefix-distr]
13:    Rule [restricted-assump-input-prefix-distl]
14:    Rule [restricted-assump-input-prefix-distr]
15:    Rule [restricted-assump-infinite-loop-distl]
16:    Rule [restricted-assump-infinite-loop-distr]
17:    Rule [restricted-assump-while-loop-distl]
18:    Rule [restricted-assump-while-loop-distr]
19:    Rule [restricted-assump-do-while-loop-distl]
20:    Rule [restricted-assump-do-while-loop-distr]
21:    Rule [restricted-assump-mid-while-loop-distl]
22:    Rule [restricted-assump-mid-while-loop-distr]
```

Rule [*operandStack*-init-*frameClass*-assump-intro]**.**

$$\begin{array}{l} \big([\mathit{arg}1?, \ldots, \mathit{arg}{<}n{>}? : \mathit{Word}; \\ \quad \mathit{stackFrame}' : \mathit{StackFrameEPC} \mid \\ \quad \langle \mathit{arg}1?, \ldots, \mathit{arg}{<}n{>}? \rangle \\ \qquad \subseteq \mathit{stackFrame}'.\mathit{localVariables} \land \\ \quad \#\,\mathit{stackFrame}'.\mathit{localVariables} = \ell \land \\ \quad \mathit{stackFrame}'.\mathit{operandStack} = \langle\rangle \land \\ \quad \mathit{stackFrame}'.\mathit{frameClass} = c \land \\ \quad \mathit{stackFrame}'.\mathit{stackSize} = s]\big) \end{array} \sqsubseteq_A \begin{array}{l} \big([\mathit{arg}1?, \ldots, \mathit{arg}{<}n{>}? : \mathit{Word}; \\ \quad \mathit{stackFrame}' : \mathit{StackFrameEPC} \mid \\ \quad \langle \mathit{arg}1?, \ldots, \mathit{arg}{<}n{>}? \rangle \\ \qquad \subseteq \mathit{stackFrame}'.\mathit{localVariables} \land \\ \quad \#\,\mathit{stackFrame}'.\mathit{localVariables} = \ell \land \\ \quad \mathit{stackFrame}'.\mathit{operandStack} = \langle\rangle \land \\ \quad \mathit{stackFrame}'.\mathit{frameClass} = c \land \\ \quad \mathit{stackFrame}'.\mathit{stackSize} = s]\big); \\ \{\#\,\mathit{stackFrame}.\mathit{operandStack} = 0\} \end{array}$$

Rule [*operandStack*-assump-unchanged-dist]**.** If A is one of

- **Skip**,
- *Poll*,
- *HandleInegSF*,

and B does not begin with $\{\# \mathit{stackFrame.operandStack} = k\}$, then

$$\{\# \mathit{stackFrame.operandStack} = k\} \ ; \ A \ ; \ B \ \sqsubseteq_A \ \begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\} \ ; \ A; \\ \{\# \mathit{stackFrame.operandStack} = k\} \ ; \ B \end{array}$$

Rule [*operandStack*-assump-increment-dist]**.** If A is one of

- *HandleAconst_nullSF*,
- *HandleDupSF*,
- *HandleAloadSF*(lvi),
- *HandleIconstSF*(n),
- $\big(\mathit{InterpreterPushSF}\big)$,

and B does not begin with $\{\# \mathit{stackFrame.operandStack} = k + 1\}$, then

$$\{\# \mathit{stackFrame.operandStack} = k\} \ ; \ A \ ; \ B \ \sqsubseteq_A \ \begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\} \ ; \ A; \\ \{\# \mathit{stackFrame.operandStack} = k + 1\} \ ; \ B \end{array}$$

Rule [*operandStack*-assump-decrement-dist]**.** If A is one of

- *HandleAstoreSF*(lvi),
- *HandleIaddSF*,
- *HandleIconstSF*(n),
- $\big(\mathit{InterpreterPopSF}\big)$,

and B does not begin with $\{\# \mathit{stackFrame.operandStack} = k - 1\}$, then

$$\{\# \mathit{stackFrame.operandStack} = k\} \ ; \ A \ ; \ B \ \sqsubseteq_A \ \begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\} \ ; \ A; \\ \{\# \mathit{stackFrame.operandStack} = k - 1\} \ ; \ B \end{array}$$

Rule [*operandStack*-assump-*InvokeSF*-dist]**.** If B does not begin with

$$\{\# \mathit{stackFrame.operandStack} = k - m\},$$

then

$$\begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\}; \\ \big(\exists \mathit{argsToPop?} == m \bullet \mathit{InvokeSF}\big) \ ; \ B \end{array} \ \sqsubseteq_A \ \begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\}; \\ \big(\exists \mathit{argsToPop?} == m \bullet \mathit{InvokeSF}\big); \\ \{\# \mathit{stackFrame.operandStack} = k - m\} \ ; \ B \end{array}$$

Rule [*HandleAconst_nullSF*-simulation]**.**

$$\begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\}; \\ \mathit{HandleAconst\_nullSF} \end{array} \ \preccurlyeq \ \mathit{stack}{<}k + 1{>} := \mathit{null}$$

Rule [*HandleDupSF*-simulation]**.**

$$\begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\}; \\ \mathit{HandleDupSF} \end{array} \ \preccurlyeq \ \mathit{stack}{<}k + 1{>} := \mathit{stack}{<}k{>}$$

Rule [*HandleAloadSF*-simulation]**.**

$$\begin{array}{l} \{\# \mathit{stackFrame.operandStack} = k\}; \\ \mathit{HandleAloadSF}(lvi) \end{array} \ \preccurlyeq \ \mathit{stack}{<}k + 1{>} := \mathit{var}{<}lvi + 1{>}$$

Rule [*HandleAstoreSF*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ HandleAstoreSF(lvi)\end{array} \preccurlyeq var{<}lvi + 1{>} := stack{<}k{>}$$

Rule [*HandleIaddSF*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ HandleIaddSF\end{array} \preccurlyeq stack{<}k - 1{>} := stack{<}k - 1{>} + stack{<}k{>}$$

Rule [*HandleIconstSF*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ HandleIconstSF(n)\end{array} \preccurlyeq stack{<}k + 1{>} := n$$

Rule [*HandleInegSF*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ HandleInegSF\end{array} \preccurlyeq stack{<}k{>} := -\, stack{<}k{>}$$

Rule [*InterpreterPopEPC*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ \big(InterpreterPopEPC[\\ \quad stackFrame / last\, frameStack,\\ \quad stackFrame' / last\, frameStack']\big)\end{array} \preccurlyeq value := stack{<}k{>}$$

Rule [*InterpreterPushEPC*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ \big(InterpreterPushEPC[\\ \quad stackFrame / last\, frameStack,\\ \quad stackFrame' / last\, frameStack']\big)\end{array} \preccurlyeq stack{<}k + 1{>} := value$$

Rule [*InterpreterPop2EPC*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ \big(InterpreterPopEPC[\\ \quad stackFrame / last\, frameStack,\\ \quad stackFrame' / last\, frameStack']\big)\end{array} \preccurlyeq \begin{array}{l}value1 := stack{<}k - 1{>};\\ value2 := stack{<}k{>}\end{array}$$

Rule [*InvokeSF*-simulation].

$$\begin{array}{l}\{\#\, stackFrame.operandStack = k\};\\ \big(\exists\, argsToPop? == m \bullet InvokeSF\big)\end{array} \preccurlyeq \begin{array}{l}poppedArgs :=\\ \quad \langle stack{<}k - m + 1{>}, \ldots, stack{<}k{>}\rangle\end{array}$$

Rule [*stackFrame*-init-simulation]**.**

$$\left(\begin{array}{l}[arg1?, \ldots, arg{<}n{>}? : Word;\\ \quad stackFrame' : StackFrameEPC \mid\\ \quad \langle arg_1, \ldots, arg_n \rangle \subseteq stackFrame'.localVariables \land\\ \quad \# stackFrame'.localVariables = \ell \land\\ \quad stackFrame'.operandStack = \langle\rangle \land\\ \quad stackFrame'.frameClass = c \land\\ \quad stackFrame'.stackSize = s]\end{array}\right) \preccurlyeq \begin{array}{l} var{<}1{>} := arg_1;\\ \quad\vdots\\ var{<}n{>} := arg_n\end{array}$$

Rule [cond-*value*1-*value*2-elim]**.**

$$\begin{array}{l}(\mathbf{var}\ value1, value2 : Word \bullet\\ \quad value1 := stack{<}k{>};\\ \quad value2 := stack{<}k+1{>};\\ \quad \mathbf{if}\ value1 \leq value2 \longrightarrow\\ \qquad \ldots\\ \quad [\!]\ value1 > value2 \longrightarrow\\ \qquad \ldots\\ \quad \mathbf{fi})\end{array} \sqsubseteq_A \begin{array}{l}\mathbf{if}\ stack{<}k{>} \leq stack{<}k+1{>} \longrightarrow\\ \quad \ldots\\ [\!]\ stack{<}k{>} > stack{<}k+1{>} \longrightarrow\\ \quad \ldots\\ \mathbf{fi}\end{array}$$

Rule [*getField*-*oid*-elim]**.**

$$\begin{array}{l}(\mathbf{var}\ oid : ObjectID \bullet\\ \quad oid := stack{<}k{>};\\ \quad getfield!oid!cid!fid\\ \quad \longrightarrow getFieldRet?value\\ \quad \longrightarrow stack{<}k{>} := value)\end{array} \sqsubseteq_A \begin{array}{l}getfield!stack{<}k{>}!cid!fid\\ \quad \longrightarrow getFieldRet?value\\ \quad \longrightarrow stack{<}k{>} := value\end{array}$$

Rule [*putField*-*oid*-*value*-elim]**.**

$$\begin{array}{l}(\mathbf{var}\ oid : ObjectID;\ value : Word \bullet\\ \quad value := stack{<}k{>};\\ \quad oid := stack{<}k-1{>};\\ \quad putField!oid!cid!fid!value\\ \quad \longrightarrow \mathbf{Skip})\end{array} \sqsubseteq_A \begin{array}{l}putField!stack{<}k-1{>}!cid!fid!stack{<}k{>}\\ \longrightarrow \mathbf{Skip}\end{array}$$

Rule [*putStatic*-*value*-elim]**.**

$$\begin{array}{l}(\mathbf{var}\ value : Word \bullet\\ \quad value := stack{<}k{>};\\ \quad putStatic!cid!fid!value \longrightarrow \mathbf{Skip})\end{array} \sqsubseteq_A putStatic!cid!fid!stack{<}k{>} \longrightarrow \mathbf{Skip}$$

Rule [*poppedArgs*-elim]**.**

$$\begin{array}{l}(\mathbf{var}\ poppedArgs : \mathrm{seq}\ Word \bullet\\ poppedArgs := \langle arg_1, \ldots, arg_n \rangle;\\ M(poppedArgs\ 1, \ldots, poppedArgs\ n))\end{array} \sqsubseteq_A M(arg_1, \ldots, arg_n)$$

Rule [*poppedArgs*-sync-elim]**.**

$$\begin{array}{lcl}
(\mathbf{var}\ \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet & & \\
\mathit{poppedArgs} := \langle \mathit{arg}_1, \ldots, \mathit{arg}_n \rangle; & & \mathit{takeLock}!\mathit{arg}_1 \\
\mathit{takeLock}!(\mathit{head}\ \mathit{methodArgs}) & \sqsubseteq_A & \longrightarrow \mathit{takeLockRet} \longrightarrow \mathbf{Skip}; \\
\longrightarrow \mathit{takeLockRet} \longrightarrow \mathbf{Skip}; & & M(\mathit{arg}_1, \ldots, \mathit{arg}_n) \\
M(\mathit{poppedArgs}\ 1, \ldots, \mathit{poppedArgs}\ n)) & &
\end{array}$$

Rule [invokevirtual-*poppedArgs*-elim]**.** If, for each $i \in 1 \mathrel{..} m$, A_i matches

$$M_i(\mathit{poppedArgs}\ 1, \ldots, \mathit{poppedArgs}\ n)$$

or

$$\begin{array}{l}
\mathit{takeLock}!(\mathit{head}\ \mathit{methodArgs}) \longrightarrow \mathit{takeLockRet} \longrightarrow \mathbf{Skip}; \\
M_i(\mathit{poppedArgs}\ 1, \ldots, \mathit{poppedArgs}\ n)
\end{array}$$

then,

$$\begin{array}{lcl}
(\mathbf{var}\ \mathit{poppedArgs} : \mathrm{seq}\ \mathit{Word} \bullet & & \\
\mathit{poppedArgs} := \langle \mathit{arg}_1, \ldots, \mathit{arg}_n \rangle; & & \mathit{getClassIDOf}!\mathit{arg}_1!\mathit{cid} \longrightarrow \\
\mathit{getClassIDOf}!(\mathit{head}\ \mathit{poppedArgs})!\mathit{cid} \longrightarrow & & \mathbf{if}\ \mathit{cid} = c_1 \longrightarrow \mathit{instantiateArgs}(A_1) \\
\mathbf{if}\ \mathit{cid} = c_1 \longrightarrow A_1 & \sqsubseteq_A & \cdots \\
\cdots & & [\!]\ \mathit{cid} = c_m \longrightarrow \mathit{instantiateArgs}(A_m) \\
[\!]\ \mathit{cid} = c_m \longrightarrow A_m & & \mathbf{fi} \\
\mathbf{fi} & &
\end{array}$$

where, for each $i \in 1 \mathrel{..} m$,

$$\begin{array}{l}
\mathit{instantiateArgs}(M_i(\mathit{poppedArgs}\ 1, \ldots, \mathit{poppedArgs}\ n)) \\
= \\
\mathit{instantiateArgs}(M_i(\mathit{arg}_1, \ldots, \mathit{arg}_n))
\end{array}$$

and

$$\begin{array}{l}
\mathit{instantiateArgs}(\mathit{takeLock}!(\mathit{head}\ \mathit{methodArgs}) \longrightarrow \mathit{takeLockRet} \longrightarrow \mathbf{Skip}; \\
\quad M_i(\mathit{poppedArgs}\ 1, \ldots, \mathit{poppedArgs}\ n)) \\
= \\
\mathit{instantiateArgs}(\mathit{takeLock}!\mathit{arg}_1 \longrightarrow \mathit{takeLockRet} \longrightarrow \mathbf{Skip}; \\
\quad M_i(\mathit{arg}_1, \ldots, \mathit{arg}_n))
\end{array}$$

Rule [*var*-parameter-conversion]**.**

$$\begin{array}{lcl}
(\mathbf{var}\ \mathit{var}1, \ldots, \mathit{var}{<}\ell{>} : \mathit{Word} \bullet & & \\
\mathbf{var}\ \mathit{stack}1, \ldots, \mathit{stack}{<}s{>} : \mathit{Word} \bullet & & (\mathbf{val}\ \mathit{var}1, \ldots \mathit{var}{<}n{>} : \mathit{Word} \bullet \\
\quad \mathit{var}1 := \mathit{arg}1; & & \mathbf{var}\ \mathit{var}{<}n+1{>}, \ldots, \mathit{var}{<}\ell{>} : \mathit{Word} \bullet \\
\quad \cdots & \sqsubseteq_A & \mathbf{var}\ \mathit{stack}1, \ldots, \mathit{stack}{<}s{>} : \mathit{Word} \bullet \\
\quad \mathit{var}{<}n{>} := \mathit{arg}{<}n{>}; & & \quad A)(\mathit{arg}1, \ldots, \mathit{arg}{<}n{>}) \\
\quad A) & &
\end{array}$$

Algorithm 20 RedefineMethodActionToExcludeParameters($methodName$)

```
1: match  $(\mathbf{val}\, var1, \ldots, var{<}n{>} : Word \bullet$          in ActionBody($methodName$) then
          $\mathbf{var}\, var{<}n+1{>}, \ldots, var{<}\ell{>}) : Word \bullet$
          $\mathbf{var}\, stack1, \ldots, stack{<}s{>} : Word \bullet$
              $A)(arg_1, \ldots, arg_n)$
2:     apply Law [action-intro]($methodName'$, $(\mathbf{val}\, var1, \ldots, var{<}n{>} : Word \bullet$     )
                                               $\mathbf{var}\, var{<}n+1{>}, \ldots, var{<}\ell{>}) : Word \bullet$
                                               $\mathbf{var}\, stack1, \ldots, stack{<}s{>} : Word \bullet$
                                                   $A)(arg_1, \ldots, arg_n)$
3: apply Law [copy-rule]($methodName'$) in reverse to ActionBody($methodName$)
4: exhaustively apply Law [copy-rule]($methodName$)
5: apply Law [action-intro]($methodName$, ActionBody($methodName$)) in reverse
6: apply Law [action-rename]($methodName'$, $methodName$)
```

Rule [argument-variable-elimination]**.** Given an action name M,

$$\begin{array}{l}(\mathbf{val}\; arg1, \ldots, arg{<}n{>} : Word \bullet \\ \quad M(arg1, \ldots, arg{<}n{>}))(arg_1, \ldots, arg_n)\end{array} \sqsubseteq_A M(arg_1, \ldots, arg_n)$$

A.3 Data Refinement of Objects

Rule [refine-*NewObject*]**.**

$$\begin{array}{l}\mathbf{var}\; thread : ThreadID;\; classID : ClassID \bullet \\ \mathbf{var}\; objectID : ObjectID;\; class : Class \bullet \\ newObject?t?c \longrightarrow thread, classID := t, c; \\ \big(GetObjectClassInfo\big); \\ AllocateObject(\\ \quad thread, sizeOfObject\; class, objectID); \\ \big(StructManObjectInit\big); \\ newObjectRet!objectID \longrightarrow \mathbf{Skip}\end{array} \sqsubseteq_A \begin{array}{l}\mathbf{var}\; objectID : ObjectID \bullet \\ newObject?thread?classID \longrightarrow \\ \mathbf{if}\; classID = {<}classID_1{>} \longrightarrow \\ \quad AllocateObject(\\ \quad\quad thread, \\ \quad\quad sizeof{<}classID_1{>}Obj, \\ \quad\quad objectID); \\ \quad \big(StructMan{<}classID_1{>}ObjInit\big); \\ \quad \vdots \\ \,[\!]\; classID = {<}classID_n{>} \longrightarrow \\ \quad AllocateObject(\\ \quad\quad thread, \\ \quad\quad sizeof{<}classID_n{>}Obj, \\ \quad\quad objectID); \\ \quad \big(StructMan{<}classID_n{>}ObjInit\big); \\ \mathbf{fi}\,;\; newObjectRet!objectID \longrightarrow \mathbf{Skip}\end{array}$$

where, for all $k \in 1 \mathrel{..} n$,

$$\begin{array}{l}\exists\, \Delta Class \mid \Xi Class \setminus (fields, fields') \bullet \\ \quad \theta\, Class = cs\; {<}classID_k{>} \land \\ \quad fields' = \bigcup\{cid : \operatorname{dom} cs \mid ({<}classID_k{>}, cid) \in subclassRel\; cs \bullet (cs\; cid).fields\} \land \\ \quad sizeof{<}classID_k{>}Obj = sizeOfObject\,(\theta\, Class\,')\end{array}$$

Rule [refine-*GetField*].

```
$\mathbf{var}\ value : Word \bullet$
$getField?objectID?classID?field \longrightarrow$
$\mathbf{if}(objectID \in \mathrm{dom}\ objects$
    $\land\ (classIDOf\ (objects\ objectID), classID) \in subclassRel\ cs) \longrightarrow$
    $\big(StructManGetField\big)\ ;\ getFieldRet!value \longrightarrow \mathbf{Skip}$
$[\!]\ (objectID \notin \mathrm{dom}\ objects$
    $\lor\ (classIDOf\ (objects\ objectID), classID) \notin subclassRel\ cs) \longrightarrow \mathbf{Chaos}$
$\mathbf{fi}$

        $\sqsubseteq_A$

$getField?oid?cid?fid \longrightarrow$
$\mathbf{if}\ oid \in \mathrm{dom}\ objects \longrightarrow$
    $\mathbf{if}\ cid = <classID_1> \land objects\ oid \in \mathrm{dom}\ cast<classID_1> \longrightarrow$
        $\mathbf{if}\ fid = <fieldID_{1,1}> \longrightarrow$
            $getFieldRet!((cast<classID_1>(objects\ oid)).<fieldID_{1,1}>) \longrightarrow \mathbf{Skip}$
            $\ldots$
            $[\!]\ fid = <fieldID_{1,m_1}> \longrightarrow$
            $getFieldRet!((cast<classID_1>(objects\ oid)).<fieldID_{1,m_1}>) \longrightarrow \mathbf{Skip}$
        $\mathbf{fi}$
        $\ldots$
    $[\!]\ cid = <classID_n> \land objects\ oid \in \mathrm{dom}\ cast<classID_n> \longrightarrow$
        $\mathbf{if}\ fid = <fieldID_{n,1}> \longrightarrow$
            $getFieldRet!((cast<classID_n>(objects\ oid)).<fieldID_{n,1}>) \longrightarrow \mathbf{Skip}$
            $\ldots$
        $[\!]\ fid = <fieldID_{n,m_n}> \longrightarrow$
            $getFieldRet!((cast<classID_n>(objects\ oid)).<fieldID_{n,m_n}>) \longrightarrow \mathbf{Skip}$
        $\mathbf{fi}$
    $\mathbf{fi}$
$[\!]\ oid \notin \mathrm{dom}\ objects \longrightarrow \mathbf{Chaos}$
$\mathbf{fi}$
```

Rule [refine-*PutField*]**.**

```
$putField?objectID?classID?field?value \longrightarrow$
$\mathbf{if}(objectID \in \mathrm{dom}\ objects$
    $\land\ (classIDOf\ (objects\ objectID), classID) \in subclassRel\ cs) \longrightarrow \big(StructManPutField\big)$
 $[]\ (objectID \notin \mathrm{dom}\ objects$
    $\lor\ (classIDOf\ (objects\ objectID), classID) \notin subclassRel\ cs) \longrightarrow \mathbf{Chaos}$
$\mathbf{fi}$

        $\sqsubseteq_A$

$putField?oid?cid?fid?value \longrightarrow$
$\mathbf{if}\ oid \in \mathrm{dom}\ objects \longrightarrow$
    $\mathbf{if}\ cid = <classID_1> \land objects\ oid \in \mathrm{dom}\ cast<classID_1> \longrightarrow$
        $\mathbf{if}\ fid = <fieldID_{1,1}> \longrightarrow$
            $objects :=$
                $objects \oplus \{oid \mapsto update<classID_1>\_<fieldID_{1,1}>\ (objects\ oid)\ value\}$
            $\ldots$
        $[]\ fid = <fieldID_{1,m_1}> \longrightarrow$
            $objects :=$
                $objects \oplus \{oid \mapsto update<classID_1>\_<fieldID_{1,m_1}>\ (objects\ oid)\ value\}$
        $\mathbf{fi}$
        $\ldots$
    $[]\ cid = <classID_n> \land objects\ oid \in \mathrm{dom}\ cast<classID_n> \longrightarrow$
        $\mathbf{if}\ fid = <fieldID_{n,1}> \longrightarrow$
            $objects :=$
                $objects \oplus \{oid \mapsto update<classID_n>\_<fieldID_{n,1}>\ (objects\ oid)\ value\}$
            $\ldots$
        $[]\ fid = <fieldID_{n,m_n}> \longrightarrow$
            $objects :=$
                $objects \oplus \{oid \mapsto update<classID_n>\_<fieldID_{n,m_n}>\ (objects\ oid)\ value\}$
        $\mathbf{fi}$
    $\mathbf{fi}$
 $[]\ oid \notin \mathrm{dom}\ objects \longrightarrow \mathbf{Chaos}$
$\mathbf{fi}$
```

Rule [refine-*GetStatic*].

```
$getStatic?cid?fid \longrightarrow$
$\mathbf{if}(cid, fid) \in \mathrm{dom}\, staticClassFields \longrightarrow$
        $\mathbf{var}\, value : Word \bullet \big(ObjManGetStatic\big);$
        $getStaticRet!value \longrightarrow \mathbf{Skip}$
 $[\!] \; (cid, fid) \notin \mathrm{dom}\, staticClassFields \longrightarrow \mathbf{Chaos}$
$\mathbf{fi}$
```

$$\sqsubseteq_A$$

```
$getStatic?cid?fid \longrightarrow$
$\mathbf{if}\; cid = {<}classID_1{>} \land fid = {<}staticFieldID_{1,1}{>} \longrightarrow$
    $getStaticRet!(staticClassFields.{<}classID_1{>}\_{<}staticFieldID_{1,1}{>}) \longrightarrow \mathbf{Skip}$
    $\ldots$
 $[\!] \; cid = {<}classID_1{>} \land fid = {<}staticFieldID_{1,\ell_1}{>} \longrightarrow$
    $getStaticRet!(staticClassFields.{<}classID_1{>}\_{<}staticFieldID_{1,\ell_1}{>}) \longrightarrow \mathbf{Skip}$
    $\ldots$
 $[\!] \; cid = {<}classID_n{>} \land fid = {<}staticFieldID_{n,1}{>} \longrightarrow$
    $getStaticRet!(staticClassFields.{<}classID_n{>}\_{<}staticFieldID_{n,1}{>}) \longrightarrow \mathbf{Skip}$
    $\ldots$
 $[\!] \; cid = {<}classID_n{>} \land fid = {<}staticFieldID_{n,\ell_n}{>} \longrightarrow$
    $getStaticRet!(staticClassFields.{<}classID_n{>}\_{<}staticFieldID_{n,\ell_n}{>}) \longrightarrow \mathbf{Skip}$
$\mathbf{fi}$
```

Rule [refine-*PutStatic*].

$$
\begin{array}{l}
\mathit{putStatic}?\mathit{cid}?\mathit{fid}?\mathit{value} \longrightarrow \\
\mathbf{if}\,(\mathit{cid},\mathit{fid}) \in \operatorname{dom} \mathit{staticClassFields} \longrightarrow \big(\mathit{StructManPutStatic}\big) \\
\;[]\; (\mathit{cid},\mathit{fid}) \in \operatorname{dom} \mathit{staticClassFields} \longrightarrow \mathbf{Chaos} \\
\mathbf{fi}
\end{array}
$$

$$\sqsubseteq_A$$

$$
\begin{array}{l}
\mathit{putStatic}?\mathit{cid}?\mathit{fid}?\mathit{value} \longrightarrow \\
\mathbf{var}\, \mathit{staticFieldsID} : \mathit{ObjectID};\; \mathit{staticFields} : \mathit{StaticFields} \bullet \\
\mathit{staticFieldsID} := (\mathit{Initialised}^{\sim})\, \mathit{staticClassFieldsID}; \\
\mathit{staticFields} := \mathit{staticClassFields}\; \mathit{staticFieldsID}; \\
\mathbf{if}\; \mathit{cid} = \mathit{<classID_1>} \land \mathit{fid} = \mathit{<staticFieldID_{1,1}>} \longrightarrow \\
\quad \mathit{staticClassFields} := \mathit{staticClassFields} \oplus \\
\quad\quad \{\mathit{staticFieldsID} \mapsto \mathit{updateStatic<classID_1>\_<staticFieldID_{1,1}>}\; \mathit{staticFields}\; \mathit{value}\} \\
\quad \ldots \\
\;[]\; \mathit{cid} = \mathit{<classID_1>} \land \mathit{fid} = \mathit{<staticFieldID_{1,\ell_1}>} \longrightarrow \\
\quad \mathit{staticClassFields} := \mathit{staticClassFields} \oplus \\
\quad\quad \{\mathit{staticFieldsID} \mapsto \mathit{updateStatic<classID_1>\_<staticFieldID_{1,\ell_1}>}\; \mathit{staticFields}\; \mathit{value}\} \\
\quad \ldots \\
\;[]\; \mathit{cid} = \mathit{<classID_n>} \land \mathit{fid} = \mathit{<staticFieldID_{n,1}>} \longrightarrow \\
\quad \mathit{staticClassFields} := \mathit{staticClassFields} \oplus \\
\quad\quad \{\mathit{staticFieldsID} \mapsto \mathit{updateStatic<classID_n>\_<staticFieldID_{n,1}>}\; \mathit{staticFields}\; \mathit{value}\} \\
\quad \ldots \\
\;[]\; \mathit{cid} = \mathit{<classID_n>} \land \mathit{fid} = \mathit{<staticFieldID_{n,\ell_n}>} \longrightarrow \\
\quad \mathit{staticClassFields} := \mathit{staticClassFields} \oplus \\
\quad\quad \{\mathit{staticFieldsID} \mapsto \mathit{updateStatic<classID_n>\_<staticFieldID_{n,\ell_n}>}\; \mathit{staticFields}\; \mathit{value}\} \\
\mathbf{fi}
\end{array}
$$

A.4 Algebraic Laws Used in the Compilation Strategy

Law [action-intro]. Given an action name N and action body B, if N is not referenced in the body of P then,

$$
\begin{array}{l}
\mathbf{process}\, P \mathrel{\widehat=} \mathbf{begin} \\
\quad \ldots \\
\quad \mathbf{state}\, S \\
\quad \ldots \\
\quad \bullet\, A \\
\quad \mathbf{end}
\end{array}
\quad = \quad
\begin{array}{l}
\mathbf{process}\, P \mathrel{\widehat=} \mathbf{begin} \\
\quad \ldots \\
\quad \mathbf{state}\, S \\
\quad \ldots \\
\quad N \mathrel{\widehat=} B \\
\quad \ldots \\
\quad \bullet\, A \\
\quad \mathbf{end}
\end{array}
$$

Law [action-rename]**.** Given action names M and N, if N is not referenced in the body of P then,

$$
\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat=} \textbf{begin} \\
\quad \ldots \\
\quad \textbf{state}\ S \\
\quad \ldots \\
\quad M \mathrel{\widehat=} B \\
\quad \ldots \\
\quad PPars \\
\quad \ldots \\
\quad \bullet\ A \\
\quad \textbf{end}
\end{array}
\quad = \quad
\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat=} \textbf{begin} \\
\quad \ldots \\
\quad \textbf{state}\ S \\
\quad \ldots \\
\quad N \mathrel{\widehat=} B \\
\quad \ldots \\
\quad PPars[N/M] \\
\quad \ldots \\
\quad \bullet\ A[N/M] \\
\quad \textbf{end}
\end{array}
$$

Law [assump-elim]**.**

$$\{g\} \sqsubseteq_A \textbf{Skip}$$

Law [copy-rule]**.** Give an action name N, if N names an action in the current process then,

$$N(e) = B(N)(e)$$

where B is a function that returns the body of an action given its name.

Law [forwards-data-refinement]**.** Given a new process state S_2 and a relation CI, if CI relates a process state S_1 to S_2, with action local state L, and, for actions A_1 and A_2,

$$\forall S_2;\ L \bullet (\exists S_1 \bullet CI),$$

and

$$\forall S_1;\ S_2;\ S_2';\ L \bullet CI \land A_2 \Rightarrow (\exists S_1';\ L' \bullet A_1 \land CI'),$$

then,

$$
\begin{array}{l}
\textbf{process}\ P1 \mathrel{\widehat=} \textbf{begin} \\
\quad \ldots \\
\quad \textbf{state}\ S_1 \\
\quad \ldots \\
\quad \bullet\ A_1 \\
\quad \textbf{end}
\end{array}
\quad \sqsubseteq_P \quad
\begin{array}{l}
\textbf{process}\ P_2 \mathrel{\widehat=} \textbf{begin} \\
\quad \ldots \\
\quad \textbf{state}\ S_2 \\
\quad \ldots \\
\quad \bullet\ A_2 \\
\quad \textbf{end}
\end{array}
$$

where A_2 is such that $A_1 \preccurlyeq A_2$

Law [process-param-elim]**.** If x is not referenced in the body of P, then

$$
\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat=} x : T \bullet \textbf{begin} \\
\quad \ldots \\
\quad \textbf{state}\ S \\
\quad \ldots \\
\quad \bullet\ A \\
\quad \textbf{end}
\end{array}
\quad = \quad
\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat=} \textbf{begin} \\
\quad \ldots \\
\quad \textbf{state}\ S \\
\quad \ldots \\
\quad \bullet\ A \\
\quad \textbf{end}
\end{array}
$$

Law [rec-action-intro]**.** Given an action B,

$$(\mu X \bullet A\,;\; X) \sqsubseteq_A (\mu X \bullet A\,;\; X)\,;\; B$$

Law [rec-rolling-rule]**.** Given action functions F and G,

$$(\mu X \bullet F(G(X))) = F(\mu X \bullet G(F(X)))$$

Law [seq-unitl]**.**

$$\mathbf{Skip}\,;\; A = A$$

Appendix B

C Code of Examples

This appendix contains the C code for the examples considered in Chapter 6. We provide the code for each example in a separate section: `PersistentSignal` in Section B.1, `Buffer` in Section B.2, and `Barrier` in Section B.3.

For each example, we first provide the Java code used as input to our prototype (Sections B.1.1, B.2.1 and B.3.1). The code input to icecap is similar, but with the addition of a file containing a main method that invokes icecap's launcher code, passing the safelet for the program. Java arrays are also used in the code input to icecap, rather than the array classes used in our code.

After the Java code for each example, we present the code generated by our prototype for each of the program methods of the examples (Sections B.1.2, B.2.2 and B.3.2). For the first example we also present the corresponding icecap code for each method. Since the icecap code is quite long and the corresponding code for each of the constructs in our prototype code is similar, we omit the icecap code for the other two examples here. It can be found among the online resources that accompany this thesis (see Section 1.4 for link).

Due to the length of some of the identifiers in the code, we have shortened the identifiers by omitting type signatures in method and field identifiers, since there are no places in the code where that would cause ambiguity. This brings the identifiers closer to those used in the corresponding icecap code. Also, since some of the lines of code are particularly long, they are broken across multiple lines in our presentation. Lines that are the continuation of a line of code in the original file are marked with a hooked arrow ($\hookrightarrow$) at the start and are not given a separate line number.

B.1 PersistentSignal

B.1.1 Java Code

B.1.1.1 MainMission.java

```java
package main;

import javax.safetycritical.Mission;


public class MainMission extends Mission {

  public long missionMemorySize() {
    return 1000000;
  }

  protected void initialize() {
    //System.out.println("Initializing main mission");

    /*
     * Signal is an AperiodicEvent with a state
     * used for backwards propagation of information
     * between the Worker and Producer
     */
    PersistentSignal signal = new PersistentSignal();

    /*
     * Create Worker APEH
     * Pass a reference to the triggering event
     * ManagedHandlers need to register themselves upon
     ↪creation
     */
    Worker worker = new Worker(signal);
    worker.register();

    /*
     * Create Producer PEH
     * Pass a reference to the event to be triggered
     * ManagedHandlers need to register themselves upon
     ↪creation
     */
    (new Producer(signal, worker, 2000, 0)).register();
  }

}
```

B.1.1.2 MainSequence.java

```java
package main;

import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;
import javax.realtime.*;
import javax.realtime.memory.ScopeParameters;


public class MainSequence extends MissionSequencer {

  public MainSequence() {
    super(
        new PriorityParameters(PriorityScheduler.
        ↪instance().getMaxPriority()),
        new ScopeParameters(
            Const.OUTERMOST_SEQ_BACKING_STORE,
            Const.PRIVATE_MEM,
            Const.IMMORTAL_MEM,
            Const.MISSION_MEM),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE)));
  }

  protected Mission getNextMission() {
    return new MainMission();
  }

}
```

B.1.1.3 MySafelet.java

```java
package main;

import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;


public class MySafelet implements Safelet {

  @Override
  public MissionSequencer getSequencer() {
    return new MainSequence();
  }

  @Override
  public long immortalMemorySize() {
    return 10000;
  }

  @Override
  public void initializeApplication() {
  }

  @Override
  public void cleanUp() {

  }

  @Override
  public long globalBackingStoreSize() {
    return 0;
  }

  @Override
  public boolean handleStartupError(int arg0, long arg1)
  ↪ {
    return false;
  }

}
```

B.1.1.4 PersistentSignal.java

```java
package main;

import javax.safetycritical.PriorityScheduler;
import javax.safetycritical.Services;

/**
 * A bivalued persistent signal
 * Used for propagation of completeness
 * information from Worker to Producer
 * @author ish503 *
 */
public class PersistentSignal {

  /*
   * Records the internal state of the signal
   */
  private boolean _set;

  public PersistentSignal(){
    super();

    /*
     * Set the ceiling priority for this shared object
     * used by Priority Ceiling Emulation protocol
     * Worker is at max priority
     */
    Services.setCeiling(this, PriorityScheduler.instance
    ↪().getMaxPriority());

    this._set = false;
  }

  /**
   * Resets the state of the signal
   */
  public synchronized void reset()
  {
    this._set = false;
  }

  /**
   * Sets the state of the signal
   */
  public synchronized void set() {
    this._set = true;
  }

  /**
   * Observes the state of the signal
   * @return true if the signal is set
   */
  public synchronized boolean isSet()
  {
    return this._set;
  }
}
```

B.1.1.5 Producer.java

```
package main;

import javax.realtime.ConfigurationParameters;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;


public class Producer extends PeriodicEventHandler {

  private PersistentSignal _signal;
  private AperiodicEventHandler _worker;

  public Producer(PersistentSignal signal,
  ↪AperiodicEventHandler worker, long period_ms, long
  ↪offset_ms) {
    super(
        new PriorityParameters(PriorityScheduler.
        ↪instance().getNormPriority()),
        new PeriodicParameters(new RelativeTime(
        ↪offset_ms, 0), new RelativeTime(period_ms, 0))
        ↪,
        new ScopeParameters(
            Const.PRIVATE_BACKING_STORE,
            Const.PRIVATE_MEM,
            0, 0),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE)));
    this._signal = signal;
    this._worker = worker;
  }

  public void handleAsyncEvent() {
    //System.out.println("\n1.1 Producer - starting
    ↪computation ");
    devices.Console.write(-11);

    /* reset signal at each release */
    this._signal.reset();
    this._worker.release();

    /* do some computation */
    //System.out.println("1.2 Producer - starting  extra
    ↪ computation ");
    devices.Console.write(-12);
    for (int i = 0; i < 1000000; i++) {
      i++;
      i--;
    }
    //System.out.println("1.3 Producer - finishing
    ↪computation ");
    devices.Console.write(-13);

    /* check if output is done */
    if (this._signal.isSet()) {
      //System.out.println("1.4 Producer - output done")
```

```
        ↪;
      devices.Console.write(-141);
    } else {
      //System.out.println("1.4 Producer - output not done yet");
      devices.Console.write(-140);
    }

  }

}
```

B.1.1.6 `Worker.java`

```
package main;

import javax.realtime.AperiodicParameters;
import javax.realtime.ConfigurationParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;


public class Worker extends AperiodicEventHandler  {

  private PersistentSignal _signal;
  private int _iteration;

  public Worker(PersistentSignal event){
    super(
        new PriorityParameters(PriorityScheduler.instance().getMaxPriority()),
        new AperiodicParameters(),
        new ScopeParameters(
            Const.PRIVATE_BACKING_STORE,
            Const.PRIVATE_MEM,
            0, 0),
        new ConfigurationParameters(-1, -1, new LongArray1(Const.HANDLER_STACK_SIZE)));

    this._signal = event;
    this._iteration = 0;
  }

  // public PersistentSignal getSignal() {  return this._signal;  }

  public void handleAsyncEvent() {
    /* do work */
    this._iteration++;
    //System.out.println("  2 Worker - output iteration: " + this._iteration + "  ");
    devices.Console.write(-2);
    devices.Console.write(this._iteration);

    /* Work done, set signal */
    this._signal.set();
  }

}
```

B.1.2 Comparison of program code

B.1.2.1 main_MySafelet_globalBackingStoreSize

B.1.2.1.1 Our code

```
void main_MySafelet_globalBackingStoreSize(int32_t var1, int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 0;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.1.2.1.2 Corresponding icecap code

There is no corresponding icecap code for this method.

B.1.2.2 `main_Producer_init`

B.1.2.2.1 Our code

```
void main_Producer_init(int32_t var1, int32_t var2, int32_t var3, int32_t var4, int32_t var5, int32_t var6, int32_t var7) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12, stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&stack4);
  if (((java_lang_Object*) ((uintptr_t)stack4))->classID == javax_safetycritical_PrioritySchedulerID) {
    javax_safetycritical_PriorityScheduler_getNormPriority(stack4, &stack4);
  }
  javax_realtime_PriorityParameters_init_I_V(stack3, stack4);
  stack3 = newObject(javax_realtime_PeriodicParametersID);
  stack4 = stack3;
  stack5 = newObject(javax_realtime_RelativeTimeID);
  stack6 = stack5;
  stack7 = var6;
  stack8 = var7;
  stack9 = 0;
  javax_realtime_RelativeTime_init(stack6, stack7, stack8, stack9);
  stack6 = newObject(javax_realtime_RelativeTimeID);
  stack7 = stack6;
  stack8 = var4;
  stack9 = var5;
  stack10 = 0;
  javax_realtime_RelativeTime_init(stack7, stack8, stack9, stack10);
  javax_realtime_PeriodicParameters_init(stack4, stack5, stack6);
  stack4 = newObject(javax_realtime_memory_ScopeParametersID);
  stack5 = stack4;
  stack6 = 0;
  stack7 = 40000;
  stack8 = 0;
  stack9 = 20000;
  stack10 = 0;
  stack11 = 0;
  stack12 = 0;
  stack13 = 0;
  javax_realtime_memory_ScopeParameters_init(stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12, stack13);
  stack5 = newObject(javax_realtime_ConfigurationParametersID);
  stack6 = stack5;
  stack7 = -1;
  stack8 = -1;
  stack9 = newObject(java_lang_LongArray1ID);
  stack10 = stack9;
  stack11 = 0;
  stack12 = 6144;
  java_lang_LongArray1_init_J_V(stack10, stack11, stack12);
  javax_realtime_ConfigurationParameters_init(stack6, stack7, stack8, stack9);
  javax_safetycritical_PeriodicEventHandler_init(stack1, stack2, stack3, stack4, stack5);
  stack1 = var1;
  stack2 = var2;
  ((main_Producer *) ((uintptr_t)stack1))->_signal = stack2;
  stack1 = var1;
  stack2 = var3;
  ((main_Producer *) ((uintptr_t)stack1))->_worker = stack2;

}
```

B.1.2.2.2 Corresponding icecap code

```
int16 main_Producer_init_(int32 *fp, int32 this, int32 signal, int32 worker, int32 period_ms, int32 lv_4, int32 offset_ms, int32 lv_6)
{
  int32* sp;
  int32 i_val12;
  int16 rval_m_5;
  int32 i_val11;
  int32 rval_5;
#if defined(JAVA_LANG_THROWABLE_INIT_)
  unsigned short pc;
#endif
  int16 excep;
  unsigned short handler_pc;
  int16 rval_m_9;
  int32 rval_9;
  int16 rval_m_13;
  int32 i_val10;
  int32 i_val9;
  int16 rval_m_28;
  int16 rval_m_38;
  int32 hvm_arg_no_3_42;
  int32 hvm_arg_no_2_42;
  int32 hvm_arg_no_1_42;
  int16 rval_m_42;
  int32 lsb_int32;
  int32 msb_int32;
  int32 i_val8;
  int32 i_val7;
  int32 i_val6;
  int32 i_val5;
  int32 i_val4;
  int16 rval_m_66;
  int16 s_val9;
  Object* narray;
  uint16 _count_;
  int8 b_val7;
  int8 index_int8;
  uint32* cobj_89;
  int16 rval_m_90;
  int32 hvm_arg_no_5_94;
  int32 hvm_arg_no_4_94;
  int32 hvm_arg_no_3_94;
  int32 hvm_arg_no_2_94;
  int32 hvm_arg_no_1_94;
  int16 rval_m_94;
  unsigned char* cobj;
  sp = &fp[9]; /* make room for local VM state on the stack */
  /*    super( */
  i_val12 = this;
  /*        new PriorityParameters(PriorityScheduler.instance().getNormPriority()), */
  *sp = (int32)i_val12;
  sp++;
  if (handleNewClassIndex(sp, 63) == 0) {
    fp[0] = *sp;
    return getClassIndex((Object*) (pointer) *sp);
  }
  sp++;
  /*        new PriorityParameters(PriorityScheduler.instance().getNormPriority()), */
  i_val12 = *(sp - 1);
  /*        new PriorityParameters(PriorityScheduler.instance().getNormPriority()), */
  sp += 1;
  rval_m_5 =
  ↪javax_safetycritical_PriorityScheduler_instance(sp);
  if (rval_m_5 == -1) {
    rval_5 = *(int32*)sp;
    i_val11 = rval_5;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_5;
  }
  sp -= 1;
  /*        new PriorityParameters(PriorityScheduler.
  ↪instance().getNormPriority()), */
  if (i_val11 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 9;
#endif
    goto throwNullPointer;
  }
  sp += 1;
  rval_m_9 =
  ↪javax_realtime_PriorityScheduler_getNormPriority(sp,
  ↪ i_val11);
  if (rval_m_9 == -1) {
    rval_9 = *(int32*)sp;
    i_val11 = rval_9;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_9;
  }
  sp -= 1;
  /*        new PriorityParameters(PriorityScheduler.
  ↪instance().getNormPriority()), */
  rval_m_13 = javax_realtime_PriorityParameters_init_(sp
  ↪, i_val12, i_val11);
  if (rval_m_13 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_13;
  }
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  if (handleNewClassIndex(sp, 91) == 0) {
    fp[0] = *sp;
    return getClassIndex((Object*) (pointer) *sp);
  }
  sp++;
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  i_val12 = *(sp - 1);
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  *sp = (int32)i_val12;
  sp++;
  if (handleNewClassIndex(sp, 133) == 0) {
    fp[0] = *sp;
    return getClassIndex((Object*) (pointer) *sp);
  }
  sp++;
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  i_val12 = *(sp - 1);
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  i_val11 = offset_ms;
  i_val10 = lv_6;
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  i_val9 = 0;
  /*        new PeriodicParameters(new RelativeTime(
  ↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
  rval_m_28 = javax_realtime_RelativeTime_init__(sp,
  ↪i_val12, i_val11, i_val10, i_val9);
  if (rval_m_28 == -1) {
    ;
```

```
}
else
{
  fp[0] = *sp;
  return rval_m_28;
}
/*          new PeriodicParameters(new RelativeTime(
↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
if (handleNewClassIndex(sp, 133) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*          new PeriodicParameters(new RelativeTime(
↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
i_val12 = *(sp - 1);
/*          new PeriodicParameters(new RelativeTime(
↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
i_val11 = period_ms;
i_val10 = lv_4;
/*          new PeriodicParameters(new RelativeTime(
↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
i_val9 = 0;
/*          new PeriodicParameters(new RelativeTime(
↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
rval_m_38 = javax_realtime_RelativeTime_init__(sp,
↪i_val12, i_val11, i_val10, i_val9);
if (rval_m_38 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_38;
}
/*          new PeriodicParameters(new RelativeTime(
↪offset_ms, 0), new RelativeTime(period_ms, 0)), */
sp--;
hvm_arg_no_3_42 = (int32)(*sp);
sp--;
hvm_arg_no_2_42 = (int32)(*sp);
sp--;
hvm_arg_no_1_42 = (int32)(*sp);
rval_m_42 = javax_realtime_PeriodicParameters_init_(sp
↪, hvm_arg_no_1_42, hvm_arg_no_2_42, hvm_arg_no_3_42)
↪;
if (rval_m_42 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_42;
}
/*          new StorageParameters( */
if (handleNewClassIndex(sp, 64) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*          new StorageParameters( */
i_val12 = *(sp - 1);
/*              Const.PRIVATE_BACKING_STORE, */
i_val11 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> PRIVATE_BACKING_STORE_f;
/*              Const.PRIVATE_BACKING_STORE, */
lsb_int32 = i_val11;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val11 = msb_int32;
i_val10 = lsb_int32;
/*              Const.PRIVATE_MEM, */
i_val9 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> PRIVATE_MEM_f;
/*              Const.PRIVATE_MEM, */
lsb_int32 = i_val9;
if (lsb_int32 < 0) {
    msb_int32 = -1;
  } else {
    msb_int32 = 0;
  }
  i_val9 = msb_int32;
  i_val8 = lsb_int32;
  /*            0, 0), */
  i_val7 = 0;
  i_val6 = 0;
  /*            0, 0), */
  i_val5 = 0;
  i_val4 = 0;
  /*        new StorageParameters( */
  rval_m_66 =
  ↪javax_safetycritical_StorageParameters_init_(sp,
  ↪i_val12, i_val11, i_val10, i_val9, i_val8, i_val7,
  ↪i_val6, i_val5, i_val4);
  if (rval_m_66 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_66;
  }
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  if (handleNewClassIndex(sp, 14) == 0) {
    fp[0] = *sp;
    return getClassIndex((Object*) (pointer) *sp);
  }
  sp++;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val12 = *(sp - 1);
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val11 = -1;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val10 = -1;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  s_val9 = 1;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  _count_ = s_val9;
  narray = (Object*) createArray(48, (uint16) _count_
  ↪FLASHARG((0)));
  if (narray == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 77;
#endif
    goto throwOutOfMemory;
  }
  i_val9 = (int32) (pointer) narray;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val8 = i_val9;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  b_val7 = 0;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val6 = ((struct _staticClassFields_c *)(pointer)
  ↪HEAP_REF((pointer)classData, staticClassFields_c*))
  ↪-> HANDLER_STACK_SIZE_f;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  lsb_int32 = i_val6;
  if (lsb_int32 < 0) {
    msb_int32 = -1;
  } else {
    msb_int32 = 0;
  }
  i_val6 = msb_int32;
  i_val5 = lsb_int32;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  lsb_int32 = i_val5;
  msb_int32 = i_val6;
  index_int8 = b_val7;
  cobj_89 = HEAP_REF((pointer)(i_val8 + sizeof(Object) +
  ↪ 2), uint32*);
  cobj_89[index_int8 << 1] = msb_int32;
  cobj_89[(index_int8 << 1) + 1] = lsb_int32;
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  rval_m_90 =
  ↪javax_realtime_ConfigurationParameters_init_(sp,
  ↪i_val12, i_val11, i_val10, i_val9);
  if (rval_m_90 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_90;
  }
  /*        new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  sp--;
  hvm_arg_no_5_94 = (int32)(*sp);
  sp--;
  hvm_arg_no_4_94 = (int32)(*sp);
  sp--;
  hvm_arg_no_3_94 = (int32)(*sp);
  sp--;
  hvm_arg_no_2_94 = (int32)(*sp);
  sp--;
  hvm_arg_no_1_94 = (int32)(*sp);
  rval_m_94 =
  ↪javax_safetycritical_PeriodicEventHandler_init_(sp,
  ↪hvm_arg_no_1_94, hvm_arg_no_2_94, hvm_arg_no_3_94,
  ↪hvm_arg_no_4_94, hvm_arg_no_5_94);
  if (rval_m_94 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_94;
  }
  /*    this._signal = signal; */
  i_val12 = this;
  /*    this._signal = signal; */
  i_val11 = signal;
  /*    this._signal = signal; */
  lsb_int32 = i_val11;
  cobj = (unsigned char *) (pointer)i_val12;
  ((struct _main_Producer_c *)HEAP_REF(cobj, void*)) ->
  ↪_signal_f = lsb_int32;
  /*    this._worker = worker; */
  i_val12 = this;
  /*    this._worker = worker; */
  i_val11 = worker;
  /*    this._worker = worker; */
  lsb_int32 = i_val11;
  cobj = (unsigned char *) (pointer)i_val12;
  ((struct _main_Producer_c *)HEAP_REF(cobj, void*)) ->
  ↪_worker_f = lsb_int32;
  /*  } */
  return -1;
  throwNullPointer:
  excep = initializeException(sp,
  ↪JAVA_LANG_NULLPOINTEREXCEPTION,
  ↪JAVA_LANG_NULLPOINTEREXCEPTION_INIT_);
  goto throwIt;
  throwOutOfMemory:
  excep = initializeException(sp,
  ↪JAVA_LANG_OUTOFMEMORYERROR,
  ↪JAVA_LANG_OUTOFMEMORYERROR_INIT_);
  goto throwIt;
  throwIt:
#if defined(JAVA_LANG_THROWABLE_INIT_)
  handler_pc = handleAthrow(& methods[531], excep, pc);
#else
  handler_pc = -1;
#endif
  sp++;
  switch(handler_pc) {
    case (unsigned short)-1: /* Not handled */
    default:
    fp[0] = *(sp - 1);
    return excep;
  }
}
```

B.1.2.3 main_PersistentSignal_reset

B.1.2.3.1 Our code

```
void main_PersistentSignal_reset(int32_t var1) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack2 = 0;
  ((main_PersistentSignal *) ((uintptr_t)stack1))->_set
  ↪= stack2;
  releaseLock(var1);
}
```

B.1.2.3.2 Corresponding icecap code

```
int16 main_PersistentSignal_reset(int32 *fp, int32 this)
{
  int32 i_val1;
  int8 b_val0;
  unsigned char* cobj;
  int8 lsb_int8;
  /*    this._set = false; */
  i_val1 = this;
  /*    this._set = false; */
  b_val0 = 0;
  /*    this._set = false; */
  lsb_int8 = b_val0;
  cobj = (unsigned char *) (pointer)i_val1;
  ((struct _main_PersistentSignal_c *)HEAP_REF(cobj,
  ↪void*)) -> _set_f = lsb_int8;
  /*  } */
  handleMonitorEnterExit((Object*)(pointer)this, 0, fp +
  ↪ 1, "");
  return -1;
}
```

B.1.2.4 main_MySafelet_cleanUp

B.1.2.4.1 Our code

```
void main_MySafelet_cleanUp(int32_t var1) {

}
```

B.1.2.4.2 Corresponding icecap code

There is no corresponding icecap code for this method.

B.1.2.5 main_PersistentSignal_isSet

B.1.2.5.1 Our code

```
void main_PersistentSignal_isSet(int32_t var1, int32_t *
 ↪ retVal) {
  int32_t stack1;
  stack1 = var1;
  stack1 = ((main_PersistentSignal *)  ((uintptr_t)
↪stack1))->_set;
  releaseLock(var1);
  *retVal = stack1;
}
```

B.1.2.5.2 Corresponding icecap code

```
int16 main_PersistentSignal_isSet(int32 *fp, int32 this)
{
  int32 i_val0;
  unsigned char* cobj;
  int8 b_val0;
  /*    return this._set; */
  i_val0 = this;
  /*    return this._set; */
  cobj = (unsigned char *) (pointer)i_val0;
  b_val0 = ((struct _main_PersistentSignal_c *)HEAP_REF(
↪cobj, void*)) -> _set_f;
  /*    return this._set; */
  handleMonitorEnterExit((Object*)(pointer)this, 0, fp +
↪ 1, "");
  return (uint8)b_val0;
}
```

B.1.2.6 main_MySafelet_handleStartupError

B.1.2.6.1 Our code

```
void main_MySafelet_handleStartupError(int32_t var1,
↪int32_t var2, int32_t var3, int32_t var4, int32_t *
↪retVal) {
  int32_t stack1;
  stack1 = 0;
  *retVal = stack1;
}
```

B.1.2.6.2 Corresponding icecap code

There is no corresponding icecap code for this method.

B.1.2.7 main_MainMission_missionMemorySize

B.1.2.7.1 Our code

```
void main_MainMission_missionMemorySize(int32_t var1,
↪int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 1000000;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.1.2.7.2 Corresponding icecap code

There is no corresponding icecap code for this method.

B.1.2.8 main_MainSequence_init

B.1.2.8.1 Our code

```
void main_MainSequence_init(int32_t var1) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->classID == javax_safetycritical_PrioritySchedulerID) {
    javax_safetycritical_PriorityScheduler_getMaxPriority(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4);
  stack3 = newObject(javax_realtime_memory_ScopeParametersID);
  stack4 = stack3;
  stack5 = 0;
  stack6 = 702000;
  stack7 = 0;
  stack8 = 20000;
  stack9 = 0;
  stack10 = 100000;
  stack11 = 0;
  stack12 = 200000;
  javax_realtime_memory_ScopeParameters_init(stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12);
  stack4 = newObject(javax_realtime_ConfigurationParametersID);
  stack5 = stack4;
  stack6 = -1;
  stack7 = -1;
  stack8 = newObject(java_lang_LongArray1ID);
  stack9 = stack8;
  stack10 = 0;
  stack11 = 6144;
  java_lang_LongArray1_init(stack9, stack10, stack11);
  javax_realtime_ConfigurationParameters_init(stack5, stack6, stack7, stack8);
  javax_safetycritical_MissionSequencer_init(stack1, stack2, stack3, stack4);

}
```

B.1.2.8.2 Corresponding icecap code

```
int16 main_MainSequence_init_(int32 *fp) {
  int32* sp;
  int32 i_val11;
  int16 rval_m_5;
  int32 i_val10;
  int32 rval_5;
#if defined(JAVA_LANG_THROWABLE_INIT_)
  unsigned short pc;
#endif
  int16 excep;
  unsigned short handler_pc;
  int16 rval_m_9;
  int32 rval_9;
  int16 rval_m_13;
  int32 lsb_int32;
  int32 msb_int32;
  int32 i_val9;
  int32 i_val8;
  int32 i_val7;
  int32 i_val6;
  int32 i_val5;
```

```
int32 i_val4;
int32 i_val3;
int16 rval_m_49;
int16 s_val8;
Object* narray;
uint16 _count_;
int8 b_val6;
int8 index_int8;
uint32* cobj_72;
int16 rval_m_73;
int32 hvm_arg_no_4_77;
int32 hvm_arg_no_3_77;
int32 hvm_arg_no_2_77;
int32 hvm_arg_no_1_77;
int16 rval_m_77;
int32
this;
this = (int32)(*(fp + 0));
sp = &fp[3]; /* make room for local VM state on the
↪stack */
/*    super( */
i_val11 = this;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
*sp = (int32) i_val11;
sp++;
if (handleNewClassIndex(sp, 63) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) * sp);
}
sp++;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
i_val11 = *(sp - 1);
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
sp += 1;
rval_m_5 =
↪javax_safetycritical_PriorityScheduler_instance(sp);
if (rval_m_5 == -1) {
  rval_5 = *(int32*) sp;
  i_val10 = rval_5;
} else {
  fp[0] = *sp;
  return rval_m_5;
}
sp -= 1;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
if (i_val10 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
  pc = 9;
#endif
  goto throwNullPointer;
}
sp += 1;
rval_m_9 =
↪javax_realtime_PriorityScheduler_getMaxPriority(sp,
↪i_val10);
if (rval_m_9 == -1) {
  rval_9 = *(int32*) sp;
  i_val10 = rval_9;
} else {
  fp[0] = *sp;
  return rval_m_9;
}
sp -= 1;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
rval_m_13 = javax_realtime_PriorityParameters_init_(sp
↪, i_val11, i_val10);
if (rval_m_13 == -1) {
  ;
} else {
  fp[0] = *sp;
  return rval_m_13;
}
/*         new StorageParameters( */
if (handleNewClassIndex(sp, 64) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) * sp);
}
sp++;
/*         new StorageParameters( */
i_val11 = *(sp - 1);
/*              Const.OUTERMOST_SEQ_BACKING_STORE, */
i_val10 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> OUTERMOST_SEQ_BACKING_STORE_f;
/*              Const.OUTERMOST_SEQ_BACKING_STORE, */
lsb_int32 = i_val10;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val10 = msb_int32;
i_val9 = lsb_int32;
/*              Const.PRIVATE_MEM, */
i_val8 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> PRIVATE_MEM_f;
/*              Const.PRIVATE_MEM, */
lsb_int32 = i_val8;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val8 = msb_int32;
i_val7 = lsb_int32;
/*              Const.IMMORTAL_MEM, */
i_val6 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> IMMORTAL_MEM_f;
/*              Const.IMMORTAL_MEM, */
lsb_int32 = i_val6;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val6 = msb_int32;
i_val5 = lsb_int32;
/*              Const.MISSION_MEM), */
i_val4 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> MISSION_MEM_f;
/*              Const.MISSION_MEM), */
lsb_int32 = i_val4;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val4 = msb_int32;
i_val3 = lsb_int32;
/*         new StorageParameters( */
rval_m_49 =
↪javax_safetycritical_StorageParameters_init_(sp,
↪i_val11,
    i_val10, i_val9, i_val8, i_val7, i_val6, i_val5,
    ↪i_val4, i_val3);
if (rval_m_49 == -1) {
  ;
} else {
  fp[0] = *sp;
  return rval_m_49;
}
/*         new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
if (handleNewClassIndex(sp, 14) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) * sp);
}
sp++;
/*         new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val11 = *(sp - 1);
/*         new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val10 = -1;
/*         new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
```

```
  i_val9 = -1;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  s_val8 = 1;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  _count_ = s_val8;
  narray = (Object*) createArray(48, (uint16) _count_
  ↪FLASHARG((0)));
  if (narray == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 60;
#endif
    goto throwOutOfMemory;
  }
  i_val8 = (int32) (pointer) narray;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val7 = i_val8;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  b_val6 = 0;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  i_val5 = ((struct _staticClassFields_c *)(pointer)
  ↪HEAP_REF((pointer)classData, staticClassFields_c*))
  ↪-> HANDLER_STACK_SIZE_f;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  lsb_int32 = i_val5;
  if (lsb_int32 < 0) {
    msb_int32 = -1;
  } else {
    msb_int32 = 0;
  }
  i_val5 = msb_int32;
  i_val4 = lsb_int32;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  lsb_int32 = i_val4;
  msb_int32 = i_val5;
  index_int8 = b_val6;
  cobj_72 = HEAP_REF((pointer)(i_val7 + sizeof(Object) +
  ↪ 2), uint32*);
  cobj_72[index_int8 << 1] = msb_int32;
  cobj_72[(index_int8 << 1) + 1] = lsb_int32;
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  rval_m_73 =
  ↪javax_realtime_ConfigurationParameters_init_(sp,
  ↪i_val11,
      i_val10, i_val9, i_val8);
  if (rval_m_73 == -1) {
    ;
  } else {
    fp[0] = *sp;
    return rval_m_73;
  }
  /*         new ConfigurationParameters(-1, -1, new long
  ↪[] {Const.HANDLER_STACK_SIZE})); */
  sp--;
  hvm_arg_no_4_77 = (int32)(*sp);
  sp--;
  hvm_arg_no_3_77 = (int32)(*sp);
  sp--;
  hvm_arg_no_2_77 = (int32)(*sp);
  sp--;
  hvm_arg_no_1_77 = (int32)(*sp);
  rval_m_77 =
  ↪javax_safetycritical_MissionSequencer_init_(sp,
  ↪hvm_arg_no_1_77,
      hvm_arg_no_2_77, hvm_arg_no_3_77, hvm_arg_no_4_77)
      ↪;
  if (rval_m_77 == -1) {
    ;
  } else {
    fp[0] = *sp;
    return rval_m_77;
  }
  /*  } */
  return -1;
  throwNullPointer: excep = initializeException(sp,
      JAVA_LANG_NULLPOINTEREXCEPTION,
      JAVA_LANG_NULLPOINTEREXCEPTION_INIT_);
  goto throwIt;
  throwOutOfMemory: excep = initializeException(sp,
      JAVA_LANG_OUTOFMEMORYERROR,
      ↪JAVA_LANG_OUTOFMEMORYERROR_INIT_);
  goto throwIt;
  throwIt:
#if defined(JAVA_LANG_THROWABLE_INIT_)
  handler_pc = handleAthrow(& methods[520], excep, pc);
#else
  handler_pc = -1;
#endif
  sp++;
  switch (handler_pc) {
  case (unsigned short) -1: /* Not handled */
  default:
    fp[0] = *(sp - 1);
    return excep;
  }
}
```

B.1.2.9 main_Producer_handleAsyncEvent

B.1.2.9.1 Our code

```
void main_Producer_handleAsyncEvent(int32_t var1) {
  int32_t var2;
  int32_t stack1, stack2;
  stack1 = -11;
  devices_Console_write(stack1);
  stack1 = var1;
  stack1 = ((main_Producer *) ((uintptr_t)stack1))->
  ↪_signal;
  if (((java_lang_Object*) ((uintptr_t)stack1))->
  ↪classID == main_PersistentSignalID) {
    takeLock(stack1);
    main_PersistentSignal_reset(stack1);
  }
  stack1 = var1;
  stack1 = ((main_Producer *) ((uintptr_t)stack1))->
  ↪_worker;
  if (((java_lang_Object*) ((uintptr_t)stack1))->
  ↪classID == main_WorkerID) {
    javax_safetycritical_AperiodicEventHandler_release(
    ↪stack1);
  }
  stack1 = -12;
  devices_Console_write(stack1);
  stack1 = 0;
  var2 = stack1;
  stack1 = var2;
  stack2 = 1000000;
  while (stack1 < stack2) {
    var2 = var2 + 1;
    var2 = var2 + -1;
    var2 = var2 + 1;
    stack1 = var2;
    stack2 = 1000000;

  }
  stack1 = -13;
  devices_Console_write(stack1);
  stack1 = var1;
  stack1 = ((main_Producer *) ((uintptr_t)stack1))->
  ↪_signal;
  if (((java_lang_Object*) ((uintptr_t)stack1))->
  ↪classID == main_PersistentSignalID) {
    takeLock(stack1);
    main_PersistentSignal_isSet(stack1, & stack1);
  }
  if (stack1 == 0) {
    stack1 = -140;
    devices_Console_write(stack1);
  } else {
    stack1 = -141;
    devices_Console_write(stack1);

  }

}
```

B.1.2.9.2 Corresponding icecap code

```
int16 main_Producer_handleAsyncEvent(int32 *fp, int32 this)
{
  int32* sp;
  int32 i_val1;
  int16 rval_m_2;
  unsigned char* cobj;
#if defined(JAVA_LANG_THROWABLE_INIT_)
  unsigned short pc;
#endif
  int16 excep;
  unsigned short handler_pc;
  int16 rval_m_13;
  int16 rval_m_24;
  int16 rval_m_30;
  int32 i_val0;
  int16 rval_m_57;
  int16 rval_m_68;
  int8 b_val1;
  int16 rval_m_78;
  int16 rval_m_88;
  int32 i;
  sp = &fp[4]; /* make room for local VM state on the stack */
  /*    devices.Console.println(-11); */
  i_val1 = (signed char)-11;
  /*    devices.Console.println(-11); */
  rval_m_2 = devices_Console_println(sp, i_val1);
  if (rval_m_2 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_2;
  }
  /*    this._signal.reset(); */
  i_val1 = this;
  /*    this._signal.reset(); */
  cobj = (unsigned char *) (pointer)i_val1;
  i_val1 = ((struct _main_Producer_c *)HEAP_REF(cobj, void*)) -> _signal_f;
  /*    this._signal.reset(); */
  if (i_val1 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 13;
#endif
    goto throwNullPointer;
  }
  handleMonitorEnterExit((Object*)(pointer)i_val1, 1, sp, "");
  rval_m_13 = main_PersistentSignal_reset(sp, i_val1);
  if (rval_m_13 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_13;
  }
  /*    this._worker.release(); */
  i_val1 = this;
  /*    this._worker.release(); */
  cobj = (unsigned char *) (pointer)i_val1;
  i_val1 = ((struct _main_Producer_c *)HEAP_REF(cobj, void*)) -> _worker_f;
  /*    this._worker.release(); */
  if (i_val1 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 24;
#endif
```

```
  goto throwNullPointer;
}
rval_m_24 =
↪javax_safetycritical_AperiodicEventHandler_release(
↪sp, i_val1);
if (rval_m_24 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_24;
}
/*    devices.Console.println(-12); */
i_val1 = (signed char)-12;
/*    devices.Console.println(-12); */
rval_m_30 = devices_Console_println(sp, i_val1);
if (rval_m_30 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_30;
}
/*    for (int i = 0; i < 1000000; i++) { */
i_val1 = 0;
/*    for (int i = 0; i < 1000000; i++) { */
i = i_val1;
/*    for (int i = 0; i < 1000000; i++) { */
goto L48;
/*      i = i + 1; */
L39:
i = (int32)i + 1;
/*      i = i - 1; */
i = (int32)i + -1;
/*    for (int i = 0; i < 1000000; i++) { */
i = (int32)i + 1;
/*    for (int i = 0; i < 1000000; i++) { */
L48:
i_val1 = (int32)i;
/*    for (int i = 0; i < 1000000; i++) { */
i_val0 = 1000000;
/*    for (int i = 0; i < 1000000; i++) { */
if (i_val1 < i_val0) {
  yieldToScheduler(sp);
  goto L39;
}
/*    devices.Console.println(-13); */
i_val1 = (signed char)-13;
/*    devices.Console.println(-13); */
rval_m_57 = devices_Console_println(sp, i_val1);
if (rval_m_57 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_57;
}
/*    if (this._signal.isSet()) { */
i_val1 = this;
/*    if (this._signal.isSet()) { */
cobj = (unsigned char *) (pointer)i_val1;
i_val1 = ((struct _main_Producer_c *)HEAP_REF(cobj,
↪void*)) -> _signal_f;
/*    if (this._signal.isSet()) { */
if (i_val1 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
  pc = 68;
#endif
  goto throwNullPointer;
}
handleMonitorEnterExit((Object*)(pointer)i_val1, 1, sp
↪, "");
rval_m_68 = main_PersistentSignal_isSet(sp, i_val1);
if (rval_m_68 >= 0) {
  b_val1 = rval_m_68;
}
else
{
  rval_m_68 = -rval_m_68;
  fp[0] = *sp;
  return rval_m_68;
}
/*    if (this._signal.isSet()) { */
if (b_val1 == 0) {
  goto L85;
}
/*      devices.Console.println(-141); */
i_val1 = -141;
/*      devices.Console.println(-141); */
rval_m_78 = devices_Console_println(sp, i_val1);
if (rval_m_78 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_78;
}
/*    } else { */
goto L92;
/*      devices.Console.println(-140); */
L85:
i_val1 = -140;
/*      devices.Console.println(-140); */
rval_m_88 = devices_Console_println(sp, i_val1);
if (rval_m_88 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_88;
}
/*  } */
L92:
return -1;
throwNullPointer:
excep = initializeException(sp,
↪JAVA_LANG_NULLPOINTEREXCEPTION,
↪JAVA_LANG_NULLPOINTEREXCEPTION_INIT_);
  goto throwIt;
  throwIt:
#if defined(JAVA_LANG_THROWABLE_INIT_)
  handler_pc = handleAthrow(& methods[532], excep, pc);
#else
  handler_pc = -1;
#endif
  sp++;
  switch(handler_pc) {
    case (unsigned short)-1: /* Not handled */
    default:
    fp[0] = *(sp - 1);
    return excep;
  }
}
```

B.1.2.10 main_Worker_init

B.1.2.10.1 Our code

```
void main_Worker_init(int32_t var1, int32_t var2) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6
    ↪, stack7, stack8, stack9, stack10, stack11, stack12,
    ↪ stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID
    ↪);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&
    ↪stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->
    ↪classID == javax_safetycritical_PrioritySchedulerID)
    ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
      ↪(stack4, & stack4);
  }
javax_realtime_PriorityParameters_init(stack3, stack4)
↪;
stack3 = newObject(
↪javax_realtime_AperiodicParametersID);
stack4 = stack3;
javax_realtime_AperiodicParameters_init(stack4);
stack4 = newObject(
↪javax_realtime_memory_ScopeParametersID);
stack5 = stack4;
stack6 = 0;
stack7 = 40000;
stack8 = 0;
stack9 = 20000;
stack10 = 0;
stack11 = 0;
stack12 = 0;
stack13 = 0;
javax_realtime_memory_ScopeParameters_init(stack5,
↪stack6, stack7, stack8, stack9, stack10, stack11,
↪stack12, stack13);
stack5 = newObject(
↪javax_realtime_ConfigurationParametersID);
stack6 = stack5;
stack7 = -1;
stack8 = -1;
stack9 = newObject(java_lang_LongArray1ID);
stack10 = stack9;
stack11 = 0;
stack12 = 6144;
java_lang_LongArray1_init_J_V(stack10, stack11,
↪stack12);
javax_realtime_ConfigurationParameters_init(stack6,
↪stack7, stack8, stack9);
javax_safetycritical_AperiodicEventHandler_init(stack1
↪, stack2, stack3, stack4, stack5);
stack1 = var1;
stack2 = var2;
((main_Worker *) ((uintptr_t)stack1))->_signal =
↪stack2;
stack1 = var1;
stack2 = 0;
((main_Worker *) ((uintptr_t)stack1))->_iteration =
↪stack2;

}
```

B.1.2.10.2 Corresponding icecap code

```
int16 main_Worker_init_(int32 *fp, int32 this, int32
↪event)
{
  int32* sp;
  int32 i_val12;
  int16 rval_m_5;
  int32 i_val11;
  int32 rval_5;
#if defined(JAVA_LANG_THROWABLE_INIT_)
  unsigned short pc;
#endif
  int16 excep;
  unsigned short handler_pc;
  int16 rval_m_9;
  int32 rval_9;
  int16 rval_m_13;
  int16 rval_m_21;
  int32 lsb_int32;
  int32 msb_int32;
  int32 i_val10;
  int32 i_val9;
  int32 i_val8;
  int32 i_val7;
  int32 i_val6;
  int32 i_val5;
  int32 i_val4;
  int16 rval_m_45;
  int16 s_val9;
  Object* narray;
  uint16 _count_;
```

```
int8 b_val7;
int8 index_int8;
uint32* cobj_68;
int16 rval_m_69;
int32 hvm_arg_no_5_73;
int32 hvm_arg_no_4_73;
int32 hvm_arg_no_3_73;
int32 hvm_arg_no_2_73;
int32 hvm_arg_no_1_73;
int16 rval_m_73;
unsigned char* cobj;
sp = &fp[4]; /* make room for local VM state on the
↪stack */
/*    super( */
i_val12 = this;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
*sp = (int32)i_val12;
sp++;
if (handleNewClassIndex(sp, 63) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
i_val12 = *(sp - 1);
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
sp += 1;
rval_m_5 =
↪javax_safetycritical_PriorityScheduler_instance(sp);
if (rval_m_5 == -1) {
  rval_5 = *(int32*)sp;
  i_val11 = rval_5;
}
else
{
  fp[0] = *sp;
  return rval_m_5;
}
sp -= 1;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
if (i_val11 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
  pc = 9;
#endif
  goto throwNullPointer;
}
sp += 1;
rval_m_9 =
↪javax_realtime_PriorityScheduler_getMaxPriority(sp,
↪i_val11);
if (rval_m_9 == -1) {
  rval_9 = *(int32*)sp;
  i_val11 = rval_9;
}
else
{
  fp[0] = *sp;
  return rval_m_9;
}
sp -= 1;
/*         new PriorityParameters(PriorityScheduler.
↪instance().getMaxPriority()), */
rval_m_13 = javax_realtime_PriorityParameters_init_(sp
↪, i_val12, i_val11);
if (rval_m_13 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_13;
}
/*         new AperiodicParameters(), */
if (handleNewClassIndex(sp, 104) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*          new AperiodicParameters(), */
i_val12 = *(sp - 1);
/*          new AperiodicParameters(), */
*sp = (int32)i_val12;
sp++;
sp -= 1;
rval_m_21 = javax_realtime_AperiodicParameters_init_(
↪sp);
if (rval_m_21 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_21;
}
/*          new StorageParameters( */
if (handleNewClassIndex(sp, 64) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*          new StorageParameters( */
i_val12 = *(sp - 1);
/*              Const.PRIVATE_BACKING_STORE, */
i_val11 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> PRIVATE_BACKING_STORE_f;
/*              Const.PRIVATE_BACKING_STORE, */
lsb_int32 = i_val11;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val11 = msb_int32;
i_val10 = lsb_int32;
/*              Const.PRIVATE_MEM, */
i_val9 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> PRIVATE_MEM_f;
/*              Const.PRIVATE_MEM, */
lsb_int32 = i_val9;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val9 = msb_int32;
i_val8 = lsb_int32;
/*              0, 0), */
i_val7 = 0;
i_val6 = 0;
/*              0, 0), */
i_val5 = 0;
i_val4 = 0;
/*          new StorageParameters( */
rval_m_45 =
↪javax_safetycritical_StorageParameters_init_(sp,
↪i_val12, i_val11, i_val10, i_val9, i_val8, i_val7,
↪i_val6, i_val5, i_val4);
if (rval_m_45 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_45;
}
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
if (handleNewClassIndex(sp, 14) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val12 = *(sp - 1);
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val11 = -1;
```

```
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val10 = -1;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
s_val9 = 1;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
_count_ = s_val9;
narray = (Object*) createArray(48, (uint16) _count_
↪FLASHARG((0)));
if (narray == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
  pc = 56;
#endif
  goto throwOutOfMemory;
}
i_val9 = (int32) (pointer) narray;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val8 = i_val9;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
b_val7 = 0;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
i_val6 = ((struct _staticClassFields_c *)(pointer)
↪HEAP_REF((pointer)classData, staticClassFields_c*))
↪-> HANDLER_STACK_SIZE_f;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
lsb_int32 = i_val6;
if (lsb_int32 < 0) {
  msb_int32 = -1;
} else {
  msb_int32 = 0;
}
i_val6 = msb_int32;
i_val5 = lsb_int32;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
lsb_int32 = i_val5;
msb_int32 = i_val6;
index_int8 = b_val7;
cobj_68 = HEAP_REF((pointer)(i_val8 + sizeof(Object) +
↪ 2), uint32*);
cobj_68[index_int8 << 1] = msb_int32;
cobj_68[(index_int8 << 1) + 1] = lsb_int32;
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
rval_m_69 =
↪javax_realtime_ConfigurationParameters_init_(sp,
↪i_val12, i_val11, i_val10, i_val9);
if (rval_m_69 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_69;
}
/*          new ConfigurationParameters(-1, -1, new long
↪[] {Const.HANDLER_STACK_SIZE})); */
sp--;
hvm_arg_no_5_73 = (int32)(*sp);
sp--;
hvm_arg_no_4_73 = (int32)(*sp);
sp--;
hvm_arg_no_3_73 = (int32)(*sp);
sp--;
hvm_arg_no_2_73 = (int32)(*sp);
sp--;
hvm_arg_no_1_73 = (int32)(*sp);
rval_m_73 =
↪javax_safetycritical_AperiodicEventHandler_init_(sp,
↪ hvm_arg_no_1_73, hvm_arg_no_2_73, hvm_arg_no_3_73,
↪hvm_arg_no_4_73, hvm_arg_no_5_73);
if (rval_m_73 == -1) {
  ;
}
else
{
    fp[0] = *sp;
    return rval_m_73;
  }
  /*    this._signal = event; */
  i_val12 = this;
  /*    this._signal = event; */
  i_val11 = event;
  /*    this._signal = event; */
  lsb_int32 = i_val11;
  cobj = (unsigned char *) (pointer)i_val12;
  ((struct _main_Worker_c *)HEAP_REF(cobj, void*)) ->
  ↪_signal_f = lsb_int32;
  /*    this._iteration = 0; */
  i_val12 = this;
  /*    this._iteration = 0; */
  i_val11 = 0;
  /*    this._iteration = 0; */
  lsb_int32 = i_val11;
  cobj = (unsigned char *) (pointer)i_val12;
  ((struct _main_Worker_c *)HEAP_REF(cobj, void*)) ->
  ↪_iteration_f = lsb_int32;
  /*  } */
  return -1;
  throwNullPointer:
  excep = initializeException(sp,
  ↪JAVA_LANG_NULLPOINTEREXCEPTION,
  ↪JAVA_LANG_NULLPOINTEREXCEPTION_INIT_);
  goto throwIt;
  throwOutOfMemory:
  excep = initializeException(sp,
  ↪JAVA_LANG_OUTOFMEMORYERROR,
  ↪JAVA_LANG_OUTOFMEMORYERROR_INIT_);
  goto throwIt;
  throwIt:
#if defined(JAVA_LANG_THROWABLE_INIT_)
  handler_pc = handleAthrow(& methods[534], excep, pc);
#else
  handler_pc = -1;
#endif
  sp++;
  switch(handler_pc) {
    case (unsigned short)-1: /* Not handled */
    default:
    fp[0] = *(sp - 1);
    return excep;
  }
}
```

B.1.2.11 main_MySafelet_getSequencer

B.1.2.11.1 Our code

```
void main_MySafelet_getSequencer(int32_t var1, int32_t *
↪ retVal) {
  int32_t stack1, stack2;
  stack1 = newObject(main_MainSequenceID);
  stack2 = stack1;
  main_MainSequence_init(stack2);
  *retVal = stack1;
}
```

B.1.2.11.2 Corresponding icecap code

```
int16 main_MySafelet_getSequencer(int32 *fp, int32 this)
{
  int32* sp;
  int32 i_val1;
  int16 rval_m_4;
  sp = &fp[3]; /* make room for local VM state on the
  ↪stack */
  /*    return new MainSequence(); */
  if (handleNewClassIndex(sp, 110) == 0) {
    fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*    return new MainSequence(); */
i_val1 = *(sp - 1);
/*    return new MainSequence(); */
*sp = (int32)i_val1;
sp++;
sp -= 1;
rval_m_4 = main_MainSequence_init_(sp);
if (rval_m_4 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_4;
}
/*    return new MainSequence(); */
sp--;
*((int32*)fp) = (int32)(*sp);
return -1;
}
```

B.1.2.12 main_PersistentSignal_init

B.1.2.12.1 Our code

```
void main_PersistentSignal_init(int32_t var1) {
  int32_t stack1, stack2;
  stack1 = var1;
  java_lang_Object_init(stack1);
  stack1 = var1;
  javax_safetycritical_PriorityScheduler_instance(&
  ↪stack2);
  if (((java_lang_Object*)  ((uintptr_t)stack2))->
  ↪classID == javax_safetycritical_PrioritySchedulerID)
  ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
    ↪(stack2, & stack2);
  }
  javax_safetycritical_Services_setCeiling(stack1,
  ↪stack2);
  stack1 = var1;
  stack2 = 0;
  ((main_PersistentSignal *) ((uintptr_t)stack1))->_set
  ↪= stack2;

}
```

B.1.2.12.2 Corresponding icecap code

```
int16 main_PersistentSignal_init_(int32 *fp) {
  int32* sp;
  int32 i_val1;
  int16 rval_m_1;
  int16 rval_m_6;
  int32 i_val0;
  int32 rval_6;
#if defined(JAVA_LANG_THROWABLE_INIT_)
  unsigned short pc;
#endif
  int16 excep;
  unsigned short handler_pc;
  int16 rval_m_10;
  int32 rval_10;
  int16 rval_m_14;
  int8 b_val0;
  unsigned char* cobj;
  int8 lsb_int8;
  int32
  this;
  this = (int32)(*(fp + 0));
  sp = & fp[3]; /* make room for local VM state on the
  ↪stack */
  /*    super(); */
  i_val1 = this;
  /*    super(); */
  *sp = (int32) i_val1;
  sp++;
  sp -= 1;
  rval_m_1 = java_lang_Object_init_(sp);
  if (rval_m_1 == -1) {
    ;
  } else {
    fp[0] = *sp;
    return rval_m_1;
  }
  /*    Services.setCeiling(this, PriorityScheduler.
  ↪instance().getMaxPriority()); */
  i_val1 = this;
  /*    Services.setCeiling(this, PriorityScheduler.
  ↪instance().getMaxPriority()); */
  sp += 1;
  rval_m_6 =
  ↪javax_safetycritical_PriorityScheduler_instance(sp);
  if (rval_m_6 == -1) {
    rval_6 = *(int32*) sp;
    i_val0 = rval_6;
  } else {
    fp[0] = *sp;
    return rval_m_6;
  }
  sp -= 1;
  /*    Services.setCeiling(this, PriorityScheduler.
  ↪instance().getMaxPriority()); */
  if (i_val0 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 10;
#endif
    goto throwNullPointer;
  }
  sp += 1;
  rval_m_10 =
  ↪javax_realtime_PriorityScheduler_getMaxPriority(sp,
  ↪i_val0);
  if (rval_m_10 == -1) {
    rval_10 = *(int32*) sp;
    i_val0 = rval_10;
  } else {
    fp[0] = *sp;
    return rval_m_10;
  }
  sp -= 1;
  /*    Services.setCeiling(this, PriorityScheduler.
  ↪instance().getMaxPriority()); */
  rval_m_14 = javax_safetycritical_Services_setCeiling(
  ↪sp, i_val1, i_val0);
  if (rval_m_14 == -1) {
    ;
  } else {
    fp[0] = *sp;
    return rval_m_14;
  }
  /*    this._set = false; */
  i_val1 = this;
  /*    this._set = false; */
  b_val0 = 0;
  /*    this._set = false; */
  lsb_int8 = b_val0;
  cobj = (unsigned char *) (pointer) i_val1;
  ((struct _main_PersistentSignal_c *)HEAP_REF(cobj,
  ↪void*)) -> _set_f = lsb_int8;
  /*  } */
  return -1;
  throwNullPointer: excep = initializeException(sp,
      JAVA_LANG_NULLPOINTEREXCEPTION,
      JAVA_LANG_NULLPOINTEREXCEPTION_INIT_);
  goto throwIt;
  throwIt:
#if defined(JAVA_LANG_THROWABLE_INIT_)
  handler_pc = handleAthrow(& methods[526], excep, pc);
#else
  handler_pc = -1;
```

```
#endif
  sp++;
  switch (handler_pc) {
  case (unsigned short) -1: /* Not handled */
  default:
    fp[0] = *(sp - 1);
    return excep;
  }
}
```

B.1.2.13 main_MainMission_init

B.1.2.13.1 Our code

```
void main_MainMission_init(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  javax_safetycritical_Mission_init(stack1);

}
```

B.1.2.13.2 Corresponding icecap code

```
int16 main_MainMission_init_(int32 *fp) {
  int32* sp;
  int32 i_val0;
  int16 rval_m_1;
  int32
  this;
  this = (int32)(*(fp + 0));
  sp = & fp[3]; /* make room for local VM state on the
  ↪stack */
  /*public class MainMission extends Mission { */
  i_val0 = this;
  /*public class MainMission extends Mission { */
  *sp = (int32) i_val0;
  sp++;
  sp -= 1;
  rval_m_1 = javax_safetycritical_Mission_init_(sp);
  if (rval_m_1 == -1) {
    ;
  } else {
    fp[0] = *sp;
    return rval_m_1;
  }
  /*public class MainMission extends Mission { */
  return -1;
}
```

B.1.2.14 main_PersistentSignal_set

B.1.2.14.1 Our code

```
void main_PersistentSignal_set(int32_t var1) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack2 = 1;
  ((main_PersistentSignal *) ((uintptr_t)stack1))->_set
  ↪= stack2;
  releaseLock(var1);
}
```

B.1.2.14.2 Corresponding icecap code

```
int16 main_PersistentSignal_set(int32 *fp, int32 this)
{
  int32 i_val1;
  int8 b_val0;
  unsigned char* cobj;
  int8 lsb_int8;
  /*    this._set = true; */
  i_val1 = this;
  /*    this._set = true; */
  b_val0 = 1;
  /*    this._set = true; */
  lsb_int8 = b_val0;
  cobj = (unsigned char *) (pointer)i_val1;
  ((struct _main_PersistentSignal_c *)HEAP_REF(cobj,
  ↪void*)) -> _set_f = lsb_int8;
  /*  } */
  handleMonitorEnterExit((Object*)(pointer)this, 0, fp +
  ↪ 1, "");
  return -1;
}
```

B.1.2.15 main_MainSequence_getNextMission

B.1.2.15.1 Our code

```
void main_MainSequence_getNextMission(int32_t var1,
↪int32_t * retVal) {
  int32_t stack1, stack2;
  stack1 = newObject(main_MainMissionID);
  stack2 = stack1;
  main_MainMission_init(stack2);
  *retVal = stack1;
}
```

B.1.2.15.2 Corresponding icecap code

```
int16 main_MainSequence_getNextMission(int32 *fp, int32
↪this)
{
  int32* sp;
  int32 i_val1;
  int16 rval_m_4;
  sp = &fp[3]; /* make room for local VM state on the
  ↪stack */
  /*    return new MainMission();    */
  if (handleNewClassIndex(sp, 73) == 0) {
    fp[0] = *sp;
    return getClassIndex((Object*) (pointer) *sp);
  }
  sp++;
  /*    return new MainMission();    */
  i_val1 = *(sp - 1);
  /*    return new MainMission();    */
  *sp = (int32)i_val1;
  sp++;
  sp -= 1;
  rval_m_4 = main_MainMission_init_(sp);
  if (rval_m_4 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_4;
  }
  /*    return new MainMission();    */
  sp--;
  *((int32*)fp) = (int32)(*sp);
  return -1;
}
```

B.1.2.16 main_MySafelet_init

B.1.2.16.1 Our code

```
void main_MySafelet_init(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  java_lang_Object_init(stack1);

}
```

B.1.2.16.2 Corresponding icecap code

```
int16 main_MySafelet_init_(int32 *fp) {
  int32* sp;
  int32 i_val0;
  int16 rval_m_1;
  int32
  this;
  this = (int32)(*(fp + 0));
  sp = & fp[3]; /* make room for local VM state on the stack */
  /*public class MySafelet implements Safelet { */
  i_val0 = this;
  /*public class MySafelet implements Safelet { */
  *sp = (int32) i_val0;
  sp++;
  sp -= 1;
  rval_m_1 = java_lang_Object_init_(sp);
  if (rval_m_1 == -1) {
    ;
  } else {
    fp[0] = *sp;
    return rval_m_1;
  }
  /*public class MySafelet implements Safelet { */
  return -1;
}
```

B.1.2.17 main_MySafelet_initializeApplication

B.1.2.17.1 Our code

```
void main_MySafelet_initializeApplication(int32_t var1) {

}
```

B.1.2.17.2 Corresponding icecap code

```
int16 main_MySafelet_initializeApplication(int32 *fp, int32 this)
{
  /*  } */
  return -1;
}
```

B.1.2.18 main_Worker_handleAsyncEvent

B.1.2.18.1 Our code

```
void main_Worker_handleAsyncEvent(int32_t var1) {
  int32_t stack1, stack2, stack3;
  stack1 = var1;
  stack2 = stack1;
  stack2 = ((main_Worker *)  ((uintptr_t)stack2))->
  ↪_iteration;
  stack3 = 1;
  stack2 = stack3 + stack2;
  ((main_Worker *) ((uintptr_t)stack1))->_iteration =
  ↪stack2;
  stack1 = -2;
  devices_Console_write(stack1);
  stack1 = var1;
  stack1 = ((main_Worker *)  ((uintptr_t)stack1))->
  ↪_iteration;
  devices_Console_write(stack1);
  stack1 = var1;
  stack1 = ((main_Worker *)  ((uintptr_t)stack1))->
  ↪_signal;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_PersistentSignalID) {
    takeLock(stack1);
    main_PersistentSignal_set(stack1);
  }

}
```

B.1.2.18.2 Corresponding icecap code

```
int16 main_Worker_handleAsyncEvent(int32 *fp, int32 this
↪)
{
  int32* sp;
  int32 i_val2;
  int32 i_val1;
  unsigned char* cobj;
  int8 b_val0;
  int8 msb_int8;
  int32 lsb_int32;
  int16 rval_m_18;
  int16 rval_m_29;
#if defined(JAVA_LANG_THROWABLE_INIT_)
  unsigned short pc;
#endif
  int16 excep;
  unsigned short handler_pc;
  int16 rval_m_40;
  sp = &fp[3]; /* make room for local VM state on the
  ↪stack */
  /*    this._iteration++; */
  i_val2 = this;
  /*    this._iteration++; */
  i_val1 = i_val2;
  /*    this._iteration++; */
  cobj = (unsigned char *) (pointer)i_val1;
  i_val1 = ((struct _main_Worker_c *)HEAP_REF(cobj, void
  ↪*)) -> _iteration_f;
  /*    this._iteration++; */
  b_val0 = 1;
  /*    this._iteration++; */
  msb_int8 = b_val0;
  lsb_int32 = i_val1;
  lsb_int32 += msb_int8;
  i_val1 = lsb_int32;
  /*    this._iteration++; */
  lsb_int32 = i_val1;
  cobj = (unsigned char *) (pointer)i_val2;
  ((struct _main_Worker_c *)HEAP_REF(cobj, void*)) ->
  ↪_iteration_f = lsb_int32;
  /*    devices.Console.println(-2); */
  i_val2 = (signed char)-2;
  /*    devices.Console.println(-2); */
  rval_m_18 = devices_Console_println(sp, i_val2);
  if (rval_m_18 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_18;
  }
  /*    devices.Console.println(this._iteration); */
  i_val2 = this;
  /*    devices.Console.println(this._iteration); */
  cobj = (unsigned char *) (pointer)i_val2;
  i_val2 = ((struct _main_Worker_c *)HEAP_REF(cobj, void
  ↪*)) -> _iteration_f;
  /*    devices.Console.println(this._iteration); */
  rval_m_29 = devices_Console_println(sp, i_val2);
  if (rval_m_29 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_29;
  }
  /*    this._signal.set();    */
  i_val2 = this;
  /*    this._signal.set();    */
  cobj = (unsigned char *) (pointer)i_val2;
  i_val2 = ((struct _main_Worker_c *)HEAP_REF(cobj, void
  ↪*)) -> _signal_f;
  /*    this._signal.set();    */
  if (i_val2 == 0) {
#if defined(JAVA_LANG_THROWABLE_INIT_)
    pc = 40;
#endif
    goto throwNullPointer;
  }
  handleMonitorEnterExit((Object*)(pointer)i_val2, 1, sp
  ↪, "");
  rval_m_40 = main_PersistentSignal_set(sp, i_val2);
  if (rval_m_40 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_40;
  }
  /*  } */
  return -1;
  throwNullPointer:
  excep = initializeException(sp,
  ↪JAVA_LANG_NULLPOINTEREXCEPTION,
  ↪JAVA_LANG_NULLPOINTEREXCEPTION_INIT_);
  goto throwIt;
  throwIt:
#if defined(JAVA_LANG_THROWABLE_INIT_)
  handler_pc = handleAthrow(&methods[535], excep, pc);
#else
  handler_pc = -1;
#endif
  sp++;
  switch(handler_pc) {
    case (unsigned short)-1: /* Not handled */
    default:
    fp[0] = *(sp - 1);
    return excep;
  }
}
```

B.1.2.19 main_MySafelet_immortalMemorySize

B.1.2.19.1 Our code

```
void main_MySafelet_immortalMemorySize(int32_t var1,
↪int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 10000;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.1.2.19.2 Corresponding icecap code

There is no corresponding icecap code for this method.

B.1.2.20 main_MainMission_initialize

B.1.2.20.1 Our code

```
void main_MainMission_initialize(int32_t var1) {
  int32_t var2, var3;
  int32_t stack1, stack2, stack3, stack4, stack5, stack6
  ↪, stack7, stack8;
  stack1 = newObject(main_PersistentSignalID);
  stack2 = stack1;
  main_PersistentSignal_init(stack2);
  var2 = stack1;
  stack1 = newObject(main_WorkerID);
  stack2 = stack1;
  stack3 = var2;
  main_Worker_init(stack2, stack3);
  var3 = stack1;
  stack1 = var3;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_WorkerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }
  stack1 = newObject(main_ProducerID);
  stack2 = stack1;
  stack3 = var2;
  stack4 = var3;
  stack5 = 0;
  stack6 = 2000;
  stack7 = 0;
  stack8 = 0;
  main_Producer_init(stack2, stack3, stack4, stack5,
  ↪stack6, stack7, stack8);
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_ProducerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }

}
```

B.1.2.20.2 Corresponding icecap code

```
int16 main_MainMission_initialize(int32 *fp, int32 this)
{
  int32* sp;
  int32 i_val7;
  int16 rval_m_4;
  int32 i_val6;
  int16 rval_m_14;
```

```
int16 rval_m_20;
int32 i_val5;
int32 msi;
int32 lsi;
const unsigned char *data_;
const ConstantInfo* constant_;
int32 i_val4;
int32 i_val3;
int32 i_val2;
int32 i_val1;
int16 rval_m_34;
int32 hvm_arg_no_1_38;
int16 rval_m_38;
int32 signal;
int32 worker;
sp = &fp[5]; /* make room for local VM state on the
↪stack */
/*    PersistentSignal signal = new PersistentSignal()
↪; */
if (handleNewClassIndex(sp, 7) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*    PersistentSignal signal = new PersistentSignal()
↪; */
i_val7 = *(sp - 1);
/*    PersistentSignal signal = new PersistentSignal()
↪; */
*sp = (int32)i_val7;
sp++;
sp -= 1;
rval_m_4 = main_PersistentSignal_init_(sp);
if (rval_m_4 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_4;
}
/*    PersistentSignal signal = new PersistentSignal()
↪; */
sp--;
signal = (int32)(*sp);
/*    Worker worker = new Worker(signal); */
if (handleNewClassIndex(sp, 69) == 0) {
  fp[0] = *sp;
  return getClassIndex((Object*) (pointer) *sp);
}
sp++;
/*    Worker worker = new Worker(signal); */
i_val7 = *(sp - 1);
/*    Worker worker = new Worker(signal); */
i_val6 = signal;
/*    Worker worker = new Worker(signal); */
rval_m_14 = main_Worker_init_(sp, i_val7, i_val6);
if (rval_m_14 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_14;
}
/*    Worker worker = new Worker(signal); */
sp--;
worker = (int32)(*sp);
/*    worker.register(); */
i_val7 = worker;
/*    worker.register(); */
rval_m_20 =
↪javax_safetycritical_AperiodicEventHandler_register(
↪sp, i_val7);
if (rval_m_20 == -1) {
  ;
}
else
{
  fp[0] = *sp;
  return rval_m_20;
}
```

```
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  if (handleNewClassIndex(sp, 92) == 0) {
    fp[0] = *sp;
    return getClassIndex((Object*) (pointer) *sp);
  }
  sp++;
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  i_val7 = *(sp - 1);
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  i_val6 = signal;
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  i_val5 = worker;
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  constant_ = &constants[96];
  data_ = (const unsigned char *) pgm_read_pointer(&constant_->data, const void **);
  msi = ((int32) pgm_read_byte(data_)) << 24;
  msi |= ((int32) pgm_read_byte(data_ +1)) << 16;
  msi |= pgm_read_byte(data_ + 2) << 8;
  msi |= pgm_read_byte(data_ + 3);
  lsi = ((int32) pgm_read_byte(data_ + 4)) << 24;
  lsi |= ((int32) pgm_read_byte(data_ + 5)) << 16;
  lsi |= pgm_read_byte(data_ + 6) << 8;
  lsi |= pgm_read_byte(data_ + 7);
  i_val4 = msi;
  i_val3 = lsi;
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  i_val2 = 0;
  i_val1 = 0;
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  rval_m_34 = main_Producer_init_(sp, i_val7, i_val6, i_val5, i_val4, i_val3, i_val2, i_val1);
  if (rval_m_34 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_34;
  }
  /*    (new Producer(signal, worker, 2000, 0)).register(); */
  sp--;
  hvm_arg_no_1_38 = (int32)(*sp);
  rval_m_38 = javax_safetycritical_PeriodicEventHandler_register(sp, hvm_arg_no_1_38);
  if (rval_m_38 == -1) {
    ;
  }
  else
  {
    fp[0] = *sp;
    return rval_m_38;
  }
  /*  } */
  return -1;
}
```

B.2 Buffer

B.2.1 Java Code

B.2.1.1 BoundedBuffer.java

```
package main;

import javax.safetycritical.PriorityScheduler;
import javax.safetycritical.Services;

public class BoundedBuffer implements Buffer {
```

```

// indices to keep track of the valid internal
↪references
private int first;
private int last;
// number of items stored
private int stored;
// maximum number of items stored
private int max = 5;
// the array to store the references
private Array<Object> data;

public BoundedBuffer() {

 /*
  * Set the ceiling priority for this shared object
  * used by Priority Ceiling Emulation protocol
  * Consumer is at max priority
  */

 Services.setCeiling(this,
    PriorityScheduler.instance().getMaxPriority())
    ↪;

   this.data = Array.<Object>newArray(this.max);
   this.first = 0;
   this.last = 0;
   this.stored = 0;
 }

 public synchronized void put(Object item) {
       // check if buffer is not full
       // Do nothing if we are already full
       if ( this.stored == this.max ) return;
       this.last = ( this.last + 1 ) % this.max
       ↪;
       this.stored++;
       //this.data[last] = item;
       this.data.store(last, item);
   }

  public synchronized Object get() {
        // check if empty
        if ( this.stored == 0 ) return null;
        this.first = (this.first + 1) % this.max
        ↪;
        this.stored--;
        //return this.data[first];
        return this.data.load(first);
    }

  public synchronized boolean isFull() { return this.
  ↪stored == this.max; }
}
```

B.2.1.2 Buffer.java

```
package main;

public interface Buffer {

 public void put(Object data);

 public Object get();

 public boolean isFull();
}
```

B.2.1.3 Consumer.java

```
package main;

import javax.realtime.AperiodicParameters;
import javax.realtime.ConfigurationParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;

import main.Buffer;
```

```

public class Consumer extends AperiodicEventHandler {

  /*
   * Reference to MissionMemory
   */
  private Buffer buffer;

  public Consumer(Buffer buffer) {

    super(//priority
        new PriorityParameters(PriorityScheduler.
        ↪instance().getMaxPriority()),
        //release
        new AperiodicParameters(),
        //storage
        new ScopeParameters(
            Const.PRIVATE_BACKING_STORE,
            Const.PRIVATE_MEM,
            0, 0),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE)));

    this.buffer = buffer;
  }

  @Override
  public void handleAsyncEvent() {
    //System.out.println("** Consumer is now handling
    ↪the 'consume' the event **");

    // System.out.println("3.1 ConsumerPrivate " + ((
    ↪ManagedMemory) RealtimeThread.getCurrentMemoryArea
    ↪()).toString());

    /*
     * Get a reference to the new object
     */
    Object data = buffer.get();

    /*
     * Confirm we can use the object
     */
    //System.out.println("3.2 Object.toString() : " + (
    ↪data.toString()) + "\n");
    devices.Console.write(data.hashCode());
  }
}
```

B.2.1.4 MainMission.java

```
package main;

import javax.safetycritical.Mission;


public class MainMission extends Mission {

  public long missionMemorySize() {
    return 1000000;
  }

  protected void initialize() {
    //System.out.println("Initializing main mission");

    /*
     * The Shared Buffer is created in MissionMemory
     */
    Buffer buffer = new BoundedBuffer();

    /*
     * Create Consumer AEH
     * Pass a reference to the shared buffer
     * ManagedHandlers need to be registered
     */
    Consumer consumer = new Consumer(buffer);
    consumer.register();

    /*
     * Create Producer PEH
     * Pass a reference to the consumer and the shared
     ↪buffer
     * ManagedHandlers need to be registered
     */
    (new Producer(consumer,buffer)).register();
  }

}
```

B.2.1.5 MainSequence.java

```
package main;

import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;
import javax.realtime.*;
import javax.realtime.memory.ScopeParameters;


public class MainSequence extends MissionSequencer {

  public MainSequence() {

    super( new PriorityParameters(PriorityScheduler.
    ↪instance().getMaxPriority()),
          new ScopeParameters(
              Const.OUTERMOST_SEQ_BACKING_STORE,
              Const.PRIVATE_MEM,
              Const.IMMORTAL_MEM,
              Const.MISSION_MEM),
          new ConfigurationParameters(-1, -1, new
          ↪LongArray1(Const.HANDLER_STACK_SIZE))
          );
  }

  protected Mission getNextMission() {
    return new MainMission();
  }

}
```

B.2.1.6 MySafelet.java

```
package main;

import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;


public class MySafelet implements Safelet {

  public MissionSequencer getSequencer() {
    return new MainSequence();
  }

  @Override
  public long immortalMemorySize() {
    return 0;
  }

  @Override
  public void initializeApplication() {

  }

  @Override
  public void cleanUp() {

  }

  @Override
  public long globalBackingStoreSize() {
    return 0;
  }

  @Override
  public boolean handleStartupError(int cause, long val)
  ↪ {
    return false;
  }

}
```

B.2.1.7 Producer.java

```
package main;

import javax.realtime.ConfigurationParameters;
import javax.realtime.MemoryArea;
import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.ManagedMemory;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;


public class Producer extends PeriodicEventHandler {
  /*
   * Event to trigger Consumer
   */
  private AperiodicEventHandler consume;

  /*
   * Keep a reference to the last object created
   */
  private Object data;

  /*
   * Limit the total number of objects to avoid running
   ↪out of memory in MissionMemory
   */
  private final int MAX_NUM_OF_OBJECTS = 5;
  private int NUM_OF_OBJECTS = 0;

  private Buffer buffer;

  private Runnable _switch = new Runnable() {
    public void run() {
      Producer.this.data = new Object();
    }
  };


// (annotations turned off to work with Java 1.4)
↪@SCJAllowed(LEVEL_1)
  public Producer(AperiodicEventHandler consumer, Buffer
  ↪ buffer) {
    /*
        //priority
        new PriorityParameters(PriorityScheduler.
        ↪instance().getNormPriority()),
        //period
        new PeriodicParameters(null, new RelativeTime
        ↪(3000, 0)),
        //storage
        new StorageParameters(32768, 4096, 4096),
        //size
        65523,
        //name
        "Producer"
    */
    super(
        new PriorityParameters(PriorityScheduler.
        ↪instance().getNormPriority()),
        new PeriodicParameters(new RelativeTime(), new
        ↪RelativeTime(3000,0)),
        new ScopeParameters(
            Const.PRIVATE_BACKING_STORE,
            Const.PRIVATE_MEM,
            0, 0),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE))
          );

    this.buffer = buffer;
    this.consume = consumer;
  }

  @Override
  public void handleAsyncEvent() {
    // System.out.println("** Producer **");
    // System.out.println("2.1 ProducerPrivate : " + ((ManagedMemory) RealtimeThread.getCurrentMemoryArea()).toString());

    /*
     * Limit the creation of new objects to avoid running out of Mission Memory
     */
    if (NUM_OF_OBJECTS <= MAX_NUM_OF_OBJECTS) {
      /*
       * Allocate new data object and update count
       */
      //try {
      ManagedMemory.executeInOuterArea(this._switch);
      //} catch (IllegalArgumentException e1) {
      //  System.out.println("2.3 Exception while trying to allocate new object in MissionMemory");
      //  System.out.println("2.3 Aborting current release");
      //  return;
      //} catch (OutOfMemoryError e1) {
      //  System.out.println("2.3 Exception while trying to allocate new object in MissionMemory");
      //  System.out.println("2.3 Aborting current release");
      //  return;
      //} catch (ExceptionInInitializerError e1) {
      //  System.out.println("2.3 Exception while trying to allocate new object in MissionMemory");
      //  System.out.println("2.3 Aborting current release");
      //  return;
      //}

      NUM_OF_OBJECTS++;

      //System.out.println("2.3 New Object[" + NUM_OF_OBJECTS +"] is in : " + (MemoryArea.getMemoryArea(data).toString()));

      /*
       * Store a reference to the new object in the buffer
       */
      this.buffer.put(data);

      /*
       * Trigger the Consumer handler
       */
      this.consume.release();
    }
  }
}
```

B.2.2 Code generated by our prototype

B.2.2.1 main_Producer_init

```
void main_Producer_init(int32_t var1, int32_t var2, int32_t var3) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12, stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->classID == javax_safetycritical_PrioritySchedulerID) {
    javax_safetycritical_PriorityScheduler_getNormPriority(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4)
  ↪;
  stack3 = newObject(javax_realtime_PeriodicParametersID
  ↪);
  stack4 = stack3;
  stack5 = newObject(javax_realtime_RelativeTimeID);
  stack6 = stack5;
  javax_realtime_RelativeTime_init(stack6);
  stack6 = newObject(javax_realtime_RelativeTimeID);
  stack7 = stack6;
  stack8 = 0;
  stack9 = 3000;
  stack10 = 0;
  javax_realtime_RelativeTime_init(stack7, stack8,
  ↪stack9, stack10);
  javax_realtime_PeriodicParameters_init(stack4, stack5,
  ↪ stack6);
  stack4 = newObject(
  ↪javax_realtime_memory_ScopeParametersID);
  stack5 = stack4;
  stack6 = 0;
  stack7 = 40000;
  stack8 = 0;
  stack9 = 20000;
  stack10 = 0;
  stack11 = 0;
  stack12 = 0;
  stack13 = 0;
  javax_realtime_memory_ScopeParameters_init(stack5,
  ↪stack6, stack7, stack8, stack9, stack10, stack11,
  ↪stack12, stack13);
  stack5 = newObject(
  ↪javax_realtime_ConfigurationParametersID);
  stack6 = stack5;
  stack7 = -1;
  stack8 = -1;
  stack9 = newObject(java_lang_LongArray1ID);
  stack10 = stack9;
  stack11 = 0;
  stack12 = 6144;
  java_lang_LongArray1_init_J_V(stack10, stack11,
  ↪stack12);
  javax_realtime_ConfigurationParameters_init(stack6,
  ↪stack7, stack8, stack9);
  javax_safetycritical_PeriodicEventHandler_init(stack1,
  ↪ stack2, stack3, stack4, stack5);
  stack1 = var1;
  stack2 = 5;
  ((main_Producer *) ((uintptr_t)stack1))->
  ↪MAX_NUM_OF_OBJECTS = stack2;
  stack1 = var1;
  stack2 = 0;
  ((main_Producer *) ((uintptr_t)stack1))->
  ↪NUM_OF_OBJECTS = stack2;
  stack1 = var1;
  stack2 = newObject(main_Producer1ID);
  stack3 = stack2;
  stack4 = var1;
  main_Producer1_init(stack3, stack4);
  ((main_Producer *) ((uintptr_t)stack1))->_switch =
  ↪stack2;
  stack1 = var1;
  stack2 = var3;
  ((main_Producer *) ((uintptr_t)stack1))->buffer =
  ↪stack2;
  stack1 = var1;
  stack2 = var2;
  ((main_Producer *) ((uintptr_t)stack1))->consume =
  ↪stack2;

}
```

B.2.2.2 main_MySafelet_globalBackingStoreSize

```
void main_MySafelet_globalBackingStoreSize(int32_t var1,
↪ int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 0;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.2.2.3 main_Producer1_run

```
void main_Producer1_run(int32_t var1) {
  int32_t stack1, stack2, stack3;
  stack1 = var1;
  stack1 = ((main_Producer1 *)  ((uintptr_t)stack1))->
  ↪this0;
  stack2 = newObject(java_lang_ObjectID);
  stack3 = stack2;
  java_lang_Object_init(stack3);
  main_Producer_access0(stack1, stack2);

}
```

B.2.2.4 main_Consumer_handleAsyncEvent

```
void main_Consumer_handleAsyncEvent(int32_t var1) {
  int32_t var2;
  int32_t stack1;
  stack1 = var1;
  stack1 = ((main_Consumer *)  ((uintptr_t)stack1))->
  ↪buffer;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_BoundedBufferID) {
    takeLock(stack1);
    main_BoundedBuffer_get(stack1, &stack1);
  }
  var2 = stack1;
  stack1 = var2;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_MainSequenceID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_ProducerID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray5ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray4ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == javax_safetycritical_io_ConsoleInputID)
  ↪ {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray1ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray3ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == javax_realtime_AperiodicParametersID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray2ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_MainMissionID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_Producer1ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_io_DataOutputStreamID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == javax_realtime_PeriodicParametersID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_Array3ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_Array2ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_Array1ID) {
    java_lang_Object_hashCode(stack1, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID ==
    ↪javax_realtime_ConfigurationParametersID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == javax_realtime_RelativeTimeID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID ==
    ↪javax_safetycritical_io_ConsoleConnectionID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == java_io_DataInputStreamID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == main_ConsumerID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == java_lang_LongArray1ID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID ==
    ↪javax_safetycritical_PrioritySchedulerID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == main_BoundedBufferID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == javax_realtime_memory_ScopeParametersID
    ↪) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == java_lang_Array5ID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == java_lang_Array4ID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == javax_safetycritical_io_ConsoleOutputID
    ↪) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == java_lang_ObjectID) {
    java_lang_Object_hashCode(stack1, & stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
    ↪->classID == javax_realtime_PriorityParametersID) {
    java_lang_Object_hashCode(stack1, & stack1);
  }
  devices_Console_write(stack1);

}
```

B.2.2.5 main_Producer_access0

```
void main_Producer_access0(int32_t var1, int32_t var2) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack2 = var2;
  ((main_Producer *) ((uintptr_t)stack1))->data = stack2
  ↪;

}
```

B.2.2.6 main_MySafelet_cleanUp

```
void main_MySafelet_cleanUp__V(int32_t var1) {

}
```

B.2.2.7 main_BoundedBuffer_put

```
void main_BoundedBuffer_put(int32_t var1, int32_t var2)
↪{
  int32_t stack1, stack2, stack3;
  stack1 = var1;
  stack1 = ((main_BoundedBuffer *)  ((uintptr_t)stack1))
  ↪->stored;
  stack2 = var1;
  stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2))
  ↪->max;
  if (stack1 != stack2) {
    stack1 = var1;
    stack2 = var1;
    stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2
    ↪))->last;
    stack3 = 1;
    stack2 = stack3 + stack2;
    stack3 = var1;
    stack3 = ((main_BoundedBuffer *)  ((uintptr_t)stack3
    ↪))->max;
    stack2 = stack3 % stack2;
    ((main_BoundedBuffer *) ((uintptr_t)stack1))->last =
    ↪ stack2;
    stack1 = var1;
    stack2 = stack1;
    stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2
    ↪))->stored;
    stack3 = 1;
    stack2 = stack3 + stack2;
    ((main_BoundedBuffer *) ((uintptr_t)stack1))->stored
    ↪ = stack2;
    stack1 = var1;
    stack1 = ((main_BoundedBuffer *)  ((uintptr_t)stack1
    ↪))->data;
    stack2 = var1;
    stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2
    ↪))->last;
    stack3 = var2;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->
    ↪classID == java_lang_Array3ID) {
      java_lang_Array3_store(stack1, stack2, stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_Array2ID) {
      java_lang_Array2_store(stack1, stack2, stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_Array1ID) {
      java_lang_Array1_store(stack1, stack2, stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_Array5ID) {
      java_lang_Array5_store(stack1, stack2, stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_Array4ID) {
      java_lang_Array4_store(stack1, stack2, stack3);
    }
    releaseLock(var1);
  } else {
    releaseLock(var1);
  }
}
```

B.2.2.8 main_Producer1_init

```
void main_Producer1_init(int32_t var1, int32_t var2) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack2 = var2;
  ((main_Producer1 *) ((uintptr_t)stack1))->this =
  ↪stack2;
  stack1 = var1;
  java_lang_Object_init(stack1);

}
```

B.2.2.9 main_MySafelet_handleStartupError

```
void main_MySafelet_handleStartupError(int32_t var1,
↪int32_t var2, int32_t var3, int32_t var4, int32_t *
↪retVal) {
  int32_t stack1;
  stack1 = 0;
  *retVal = stack1;
}
```

B.2.2.10 main_MainMission_missionMemorySize

```
void main_MainMission_missionMemorySize(int32_t var1,
↪int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 1000000;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.2.2.11 main_MainSequence_init

```
void main_MainSequence_init(int32_t var1) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6
  ↪, stack7, stack8, stack9, stack10, stack11, stack12;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID
  ↪);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&
  ↪stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->
  ↪classID == javax_safetycritical_PrioritySchedulerID)
  ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
    ↪(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4)
  ↪;
  stack3 = newObject(
  ↪javax_realtime_memory_ScopeParametersID);
  stack4 = stack3;
  stack5 = 0;
  stack6 = 702000;
  stack7 = 0;
  stack8 = 20000;
  stack9 = 0;
  stack10 = 100000;
  stack11 = 0;
  stack12 = 200000;
  javax_realtime_memory_ScopeParameters_init(stack4,
  ↪stack5, stack6, stack7, stack8, stack9, stack10,
  ↪stack11, stack12);
  stack4 = newObject(
  ↪javax_realtime_ConfigurationParametersID);
  stack5 = stack4;
  stack6 = -1;
  stack7 = -1;
  stack8 = newObject(java_lang_LongArray1ID);
  stack9 = stack8;
  stack10 = 0;
  stack11 = 6144;
  java_lang_LongArray1_init(stack9, stack10, stack11);
  javax_realtime_ConfigurationParameters_init(stack5,
  ↪stack6, stack7, stack8);
  javax_safetycritical_MissionSequencer_init(stack1,
  ↪stack2, stack3, stack4);

}
```

B.2.2.12 main_Producer_handleAsyncEvent

```
void main_Producer_handleAsyncEvent(int32_t var1) {
  int32_t stack1, stack2, stack3;
  stack1 = var1;
  stack1 = ((main_Producer *)  ((uintptr_t)stack1))->
  ↪NUM_OF_OBJECTS;
  stack2 = 5;
  if (stack1 > stack2) {

  } else {
    stack1 = var1;
    stack1 = ((main_Producer *)  ((uintptr_t)stack1))->
    ↪_switch;
    javax_safetycritical_ManagedMemory_executeInOuterArea
    ↪(stack1);
    stack1 = var1;
    stack2 = stack1;
    stack2 = ((main_Producer *)  ((uintptr_t)stack2))->
    ↪NUM_OF_OBJECTS;
    stack3 = 1;
    stack2 = stack3 + stack2;
    ((main_Producer *) ((uintptr_t)stack1))->
    ↪NUM_OF_OBJECTS = stack2;
    stack1 = var1;
    stack1 = ((main_Producer *)  ((uintptr_t)stack1))->
    ↪buffer;
    stack2 = var1;
    stack2 = ((main_Producer *)  ((uintptr_t)stack2))->
    ↪data;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->
    ↪classID == main_BoundedBufferID) {
      takeLock(stack1);
      main_BoundedBuffer_put(stack1, stack2);
    }
    stack1 = var1;
    stack1 = ((main_Producer *)  ((uintptr_t)stack1))->
    ↪consume;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->
    ↪classID == main_ConsumerID) {
      javax_safetycritical_AperiodicEventHandler_release
      ↪(stack1);
    }

  }
}
```

B.2.2.13 main_MySafelet_getSequencer

```
void main_MySafelet_getSequencer(int32_t var1, int32_t *
↪ retVal) {
  int32_t stack1, stack2;
  stack1 = newObject(main_MainSequenceID);
  stack2 = stack1;
  main_MainSequence_init(stack2);
  *retVal = stack1;
}
```

B.2.2.14 main_BoundedBuffer_init

```
void main_BoundedBuffer_init(int32_t var1) {
  int32_t stack1, stack2;
  stack1 = var1;
  java_lang_Object_init(stack1);
  stack1 = var1;
  stack2 = 5;
  ((main_BoundedBuffer *) ((uintptr_t)stack1))->max =
  ↪stack2;
  stack1 = var1;
  javax_safetycritical_PriorityScheduler_instance(&
  ↪stack2);
  if (((java_lang_Object*)  ((uintptr_t)stack2))->
  ↪classID == javax_safetycritical_PrioritySchedulerID)
  ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
    ↪(stack2, & stack2);
  }
  javax_safetycritical_Services_setCeiling(stack1,
  ↪stack2);
  stack1 = var1;
  stack2 = var1;
  stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2))
  ↪->max;
  java_lang_Array_newArray(stack2, & stack2);
  ((main_BoundedBuffer *) ((uintptr_t)stack1))->data =
  ↪stack2;
  stack1 = var1;
  stack2 = 0;
  ((main_BoundedBuffer *) ((uintptr_t)stack1))->first =
  ↪stack2;
  stack1 = var1;
  stack2 = 0;
  ((main_BoundedBuffer *) ((uintptr_t)stack1))->last =
  ↪stack2;
  stack1 = var1;
  stack2 = 0;
  ((main_BoundedBuffer *) ((uintptr_t)stack1))->stored =
  ↪ stack2;

}
```

B.2.2.15 main_BoundedBuffer_isFull

```
void main_BoundedBuffer_isFull(int32_t var1, int32_t *
↪retVal) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack1 = ((main_BoundedBuffer *)  ((uintptr_t)stack1))
  ↪->stored;
  stack2 = var1;
  stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2))
  ↪->max;
  if (stack1 != stack2) {
    stack1 = 0;
    releaseLock(var1);
  } else {
    stack1 = 1;
    releaseLock(var1);
  }
  *retVal = stack1;
}
```

B.2.2.16 main_MainMission_init

```
void main_MainMission_init(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  javax_safetycritical_Mission_init(stack1);

}
```

B.2.2.17 main_MainSequence_getNextMission

```
void main_MainSequence_getNextMission(int32_t var1,
↪int32_t * retVal) {
  int32_t stack1, stack2;
  stack1 = newObject(main_MainMissionID);
  stack2 = stack1;
  main_MainMission_init(stack2);
  *retVal = stack1;
}
```

B.2.2.18 main_MySafelet_init

```
void main_MySafelet_init(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  java_lang_Object_init(stack1);

}
```

B.2.2.19 main_BoundedBuffer_get

```
void main_BoundedBuffer_get(int32_t var1, int32_t *
↪retVal) {
  int32_t stack1, stack2, stack3;
  stack1 = var1;
  stack1 = ((main_BoundedBuffer *)  ((uintptr_t)stack1))
  ↪->stored;
  if (stack1 != 0) {
    stack1 = var1;
    stack2 = var1;
    stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2
    ↪))->first;
    stack3 = 1;
    stack2 = stack3 + stack2;
    stack3 = var1;
    stack3 = ((main_BoundedBuffer *)  ((uintptr_t)stack3
    ↪))->max;
    stack2 = stack3 % stack2;
    ((main_BoundedBuffer *) ((uintptr_t)stack1))->first
    ↪= stack2;
    stack1 = var1;
    stack2 = stack1;
    stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2
    ↪))->stored;
    stack3 = 1;
    stack2 = stack3 - stack2;
    ((main_BoundedBuffer *) ((uintptr_t)stack1))->stored
    ↪ = stack2;
    stack1 = var1;
    stack1 = ((main_BoundedBuffer *)  ((uintptr_t)stack1))->data;
    stack2 = var1;
    stack2 = ((main_BoundedBuffer *)  ((uintptr_t)stack2))->first;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_Array3ID) {
      java_lang_Array3_load(stack1, stack2, & stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_Array2ID) {
      java_lang_Array2_load(stack1, stack2, & stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_Array1ID) {
      java_lang_Array1_load(stack1, stack2, & stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_Array5ID) {
      java_lang_Array5_load(stack1, stack2, & stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_Array4ID) {
      java_lang_Array4_load(stack1, stack2, & stack1);
    }
    releaseLock(var1);
  } else {
    stack1 = 0;
    releaseLock(var1);
  }
  *retVal = stack1;
}
```

B.2.2.20 main_MySafelet_initializeApplication

```
void main_MySafelet_initializeApplication(int32_t var1){

}
```

B.2.2.21 main_Consumer_init

```
void main_Consumer_init(int32_t var1, int32_t var2) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12, stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->classID == javax_safetycritical_PrioritySchedulerID) {
    javax_safetycritical_PriorityScheduler_getMaxPriority(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4);
  stack3 = newObject(javax_realtime_AperiodicParametersID);
  stack4 = stack3;
  javax_realtime_AperiodicParameters_init(stack4);
  stack4 = newObject(javax_realtime_memory_ScopeParametersID);
  stack5 = stack4;
  stack6 = 0;
  stack7 = 40000;
  stack8 = 0;
  stack9 = 20000;
  stack10 = 0;
  stack11 = 0;
  stack12 = 0;
  stack13 = 0;
  javax_realtime_memory_ScopeParameters_init(stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12, stack13);
  stack5 = newObject(javax_realtime_ConfigurationParametersID);
  stack6 = stack5;
  stack7 = -1;
  stack8 = -1;
  stack9 = newObject(java_lang_LongArray1ID);
  stack10 = stack9;
  stack11 = 0;
  stack12 = 6144;
  java_lang_LongArray1_init(stack10, stack11, stack12);
  javax_realtime_ConfigurationParameters_init(stack6,
  ↪stack7, stack8, stack9);
  javax_safetycritical_AperiodicEventHandler_init(stack1
  ↪, stack2, stack3, stack4, stack5);
  stack1 = var1;
  stack2 = var2;
  ((main_Consumer *) ((uintptr_t)stack1))->buffer =
  ↪stack2;

}
```

B.2.2.22 main_MySafelet_immortalMemorySize

```
void main_MySafelet_immortalMemorySize(int32_t var1,
↪int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 0;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.2.2.23 main_MainMission_initialize

```
void main_MainMission_initialize(int32_t var1) {
  int32_t var2, var3;
  int32_t stack1, stack2, stack3, stack4;
  stack1 = newObject(main_BoundedBufferID);
  stack2 = stack1;
  main_BoundedBuffer_init(stack2);
  var2 = stack1;
  stack1 = newObject(main_ConsumerID);
  stack2 = stack1;
  stack3 = var2;
  main_Consumer_init(stack2, stack3);
  var3 = stack1;
  stack1 = var3;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_ConsumerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }
  stack1 = newObject(main_ProducerID);
  stack2 = stack1;
  stack3 = var3;
  stack4 = var2;
  main_Producer_init(stack2, stack3, stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_ProducerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }

}
```

B.3 Barrier

B.3.1 Java Code

B.3.1.1 Barrier.java

```
package main;

import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.safetycritical.Services;

/**
 * Controls the synchronisation between several handlers
 * Each handler must trigger the barrier with its own
↪unique id
 * When all have done so, the Aperiodic Event
```

```java
 * passed during initialisation is fired
 * @author ish503
 *
 */
public class Barrier {
  private BooleanArray flag;
  private AperiodicEventHandler e;

  /**
   * Creates a Barrier
   * @param size the number of handlers of interest
   * @param launch the event to be called when the
   ↪barrier is released
   */
  public Barrier(int size, AperiodicEventHandler launch)
  ↪ {
    /*
     * Set the ceiling priority for this shared object
     * used by Priority Ceiling Emulation protocol
     * FireHandler is at max priority
     */
    Services.setCeiling(this,
        PriorityScheduler.instance().getMaxPriority());

    this.flag = BooleanArray.newArray(size);
    this.e = launch;
  }

  /**
   * Checks if all handlers have triggered the barrier
   * @return true if all handlers have triggered the
   ↪barrier
   */
  public synchronized boolean isOkToFire() {
    boolean okToFire = true;

    for (int i = 0; i < this.flag.length(); i++) {
      if (this.flag.load(i) == false) {
        okToFire = false;
      }
    }

    return okToFire;
  }

  /**
   * Triggers the barrier for the specified handler id
   * @param id
   */
  public synchronized void trigger(int id) {
    this.flag.store(id, true);

    if (isOkToFire())
      {
        this.e.release();
        this.reset();
      }
  }

  /**
   * Checks if the handler has already triggered the
   ↪barrier
   * @param id the unique handler id
   * @return true if the handler has already triggered
   ↪the barrier
   */
  public synchronized boolean isAlreadyTriggered(int id)
  ↪ {
    return this.flag.load(id);
  }

  /**
   * Resets the barrier.
   * The event to be fired during the next barrier
   ↪release
   * is not changed.
   */
  private synchronized void reset() {
    for (int i = 0; i < this.flag.length(); i++)
    {
      this.flag.store(i, false);
    }
  }
}
```

B.3.1.2 Button.java

```
package main;

import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.safetycritical.Services;

/**
 * Controls the synchronisation between several handlers
 * Each handler must trigger the barrier with its own unique id
 * When all have done so, the Aperiodic Event
 * passed during initialisation is fired
 * @author ish503
 *
 */
public class Barrier {
  private BooleanArray flag;
  private AperiodicEventHandler e;

  /**
   * Creates a Barrier
   * @param size the number of handlers of interest
   * @param launch the event to be called when the barrier is released
   */
  public Barrier(int size, AperiodicEventHandler launch) {
    /*
     * Set the ceiling priority for this shared object
     * used by Priority Ceiling Emulation protocol
     * FireHandler is at max priority
     */
    Services.setCeiling(this,
        PriorityScheduler.instance().getMaxPriority());

    this.flag = BooleanArray.newArray(size);
    this.e = launch;
  }

  /**
   * Checks if all handlers have triggered the barrier
   * @return true if all handlers have triggered the barrier
   */
  public synchronized boolean isOkToFire() {
    boolean okToFire = true;

    for (int i = 0; i < this.flag.length(); i++) {
      if (this.flag.load(i) == false) {
        okToFire = false;
      }
    }

    return okToFire;
  }

  /**
   * Triggers the barrier for the specified handler id
   * @param id
   */
  public synchronized void trigger(int id) {
    this.flag.store(id, true);

    if (isOkToFire())
      {
        this.e.release();
        this.reset();
      }
  }

  /**
   * Checks if the handler has already triggered the barrier
   * @param id the unique handler id
   * @return true if the handler has already triggered the barrier
   */
  public synchronized boolean isAlreadyTriggered(int id)
  ↪ {
    return this.flag.load(id);
  }

  /**
   * Resets the barrier.
   * The event to be fired during the next barrier
   ↪release
   * is not changed.
   */
  private synchronized void reset() {
    for (int i = 0; i < this.flag.length(); i++)
    {
      this.flag.store(i, false);
    }
  }
}
```

B.3.1.3 FireHandler.java

```
package main;

import javax.realtime.AperiodicParameters;
import javax.realtime.ConfigurationParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;

public class FireHandler extends AperiodicEventHandler {

  /*
   * Reference to MissionMemory
   */
  private Barrier barrier;
  private int id;

  public FireHandler(Barrier barrier, int id) {

    super(
        new PriorityParameters(PriorityScheduler.
        ↪instance().getMaxPriority()),
        new AperiodicParameters(),
        new ScopeParameters(
            Const.PRIVATE_BACKING_STORE,
            Const.PRIVATE_MEM,
            0, 0),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE)));

    this.barrier = barrier;
    this.id = id;
  }

  public void handleAsyncEvent() {
      //devices.Console.println("\n** FireHandler is now
      ↪ handling event " + id + " **");
      devices.Console.write(id);

      /*
       * If we have already triggered the barrier,
       * do not retrigger
       */
      if (barrier.isAlreadyTriggered(this.id)) return;

      barrier.trigger(this.id);
  }

}
```

B.3.1.4 LaunchHandler.java

```
package main;

import javax.realtime.AperiodicParameters;
import javax.realtime.ConfigurationParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;

public class LaunchHandler extends AperiodicEventHandler
↪ {

  public LaunchHandler() {

    super(
        new PriorityParameters(PriorityScheduler.
        ↪instance().getMaxPriority()),
        new AperiodicParameters(),
        new ScopeParameters(
            Const.PRIVATE_BACKING_STORE,
            Const.PRIVATE_MEM,
            0, 0),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE)));
  }

  public void handleAsyncEvent() {
      //devices.Console.println(" LAUNCHING MISSILE ");
      devices.Console.write(-1);
  }

}
```

B.3.1.5 MainMission.java

```
package main;

import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.Mission;


public class MainMission extends Mission {

  public long missionMemorySize() {
    return 1000000;
  }

  public void initialize() {
    //devices.Console.println("Initializing main mission
    ↪");

    /*
     * Create Launch AEH
     * Pass a reference to the shared barrier
     * ManagedHandlers need to register themselves upon
     ↪creation
     */
    AperiodicEventHandler launch = new LaunchHandler();
    launch.register();

    /* Create a barrier for 2 handlers,
     * Triggers launch event when ready to proceed
     */
    Barrier barrier = new Barrier(2, launch);

    /* The fire1 and fire2 events release fire1Handler
    ↪and fire2Handler. */
    AperiodicEventHandler fire1 = new FireHandler(
    ↪barrier, 0);
    AperiodicEventHandler fire2 = new FireHandler(
    ↪barrier, 1);

    /*
     * Create Fire1 and Fire2 AEH
     * Pass a reference to the shared barrier
     * ManagedHandlers need to register themselves upon
     ↪creation
     */
    fire1.register();
    fire2.register();

    /*
     * Create PEHs that generate event occurrences.
     */
    (new Button(fire1, 2000, 0)).register();  //2s
    (new Button(fire2, 9000, 9000)).register(); //9s + 9
    ↪s offset
  }
}
```

B.3.1.6 MainSequence.java

```
package main;

import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.PriorityScheduler;
import javax.scj.util.Const;
import javax.realtime.*;
import javax.realtime.memory.ScopeParameters;


public class MainSequence extends MissionSequencer {

  public MainSequence() {
    super(
        new PriorityParameters(PriorityScheduler.
        ↪instance().getMaxPriority()),
        new ScopeParameters(
            Const.OUTERMOST_SEQ_BACKING_STORE,
            Const.PRIVATE_MEM,
            Const.IMMORTAL_MEM,
            Const.MISSION_MEM),
        new ConfigurationParameters(-1, -1, new
        ↪LongArray1(Const.HANDLER_STACK_SIZE)));
  }

  protected Mission getNextMission() {
    return new MainMission();
  }

}
```

B.3.1.7 MySafelet.java

```
package main;

import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;

public class MySafelet implements Safelet {

  @Override
  public MissionSequencer getSequencer() {
    return new MainSequence();
  }

  @Override
  public long immortalMemorySize() {
    return 0;
  }


  @Override
  public void initializeApplication() {

  }

  @Override
  public void cleanUp() {

  }

  @Override
  public long globalBackingStoreSize() {
    return 0;
  }

  @Override
  public boolean handleStartupError(int cause, long val)
  ↪ {
    return false;
  }

}
```

B.3.2 Code generated by our prototype

B.3.2.1 main_MySafelet_globalBackingStoreSize

```
void main_MySafelet_globalBackingStoreSize(int32_t var1, int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 0;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.3.2.2 main_LaunchHandler_handleAsyncEvent

```
void main_LaunchHandler_handleAsyncEvent(int32_t var1) {
  int32_t stack1;
  stack1 = -1;
  devices_Console_write(stack1);

}
```

B.3.2.3 main_Barrier_isOkToFire

```
void main_Barrier_isOkToFire(int32_t var1, int32_t *retVal) {
  int32_t var2, var3;
  int32_t stack1, stack2;
  stack1 = 1;
  var2 = stack1;
  stack1 = 0;
  var3 = stack1;
  stack1 = var3;
  stack2 = var1;
  stack2 = ((main_Barrier *)  ((uintptr_t)stack2))->flag;
  if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray5ID) {
    java_lang_BooleanArray5_length(stack2, & stack2);
  } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray4ID) {
    java_lang_BooleanArray4_length(stack2, & stack2);
  } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray1ID) {
    java_lang_BooleanArray1_length(stack2, & stack2);
  } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray3ID) {
    java_lang_BooleanArray3_length(stack2, & stack2);
  } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray2ID) {
    java_lang_BooleanArray2_length(stack2, & stack2);
  }
  while (stack1 < stack2) {
    stack1 = var1;
    stack1 = ((main_Barrier *)  ((uintptr_t)stack1))->flag;
    stack2 = var3;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray5ID) {
      java_lang_BooleanArray5_load(stack1, stack2, &stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray4ID) {
      java_lang_BooleanArray4_load(stack1, stack2, &stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray1ID) {
      java_lang_BooleanArray1_load(stack1, stack2, &stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray3ID) {
      java_lang_BooleanArray3_load(stack1, stack2, &stack1);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray2ID) {
      java_lang_BooleanArray2_load(stack1, stack2, &stack1);
    }
    if (!(stack1 != 0)) {
      stack1 = 0;
      var2 = stack1;
    }
    var3 = var3 + 1;
    stack1 = var3;
    stack2 = var1;
    stack2 = ((main_Barrier *)  ((uintptr_t)stack2))->flagLjava_lang_BooleanArray_;
    if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray5ID) {
      java_lang_BooleanArray5_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray4ID) {
      java_lang_BooleanArray4_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray1ID) {
      java_lang_BooleanArray1_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray3ID) {
      java_lang_BooleanArray3_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2))->classID == java_lang_BooleanArray2ID) {
      java_lang_BooleanArray2_length(stack2, &stack2);
    }

  }
  stack1 = var2;
  releaseLock(var1);
  *retVal = stack1;
}
```

B.3.2.4 main_MySafelet_cleanUp

```
void main_MySafelet_cleanUp(int32_t var1) {

}
```

B.3.2.5 main_MySafelet_handleStartupError

```
void main_MySafelet_handleStartupError(int32_t var1, int32_t var2, int32_t var3, int32_t var4, int32_t * retVal) {
  int32_t stack1;
  stack1 = 0;
  *retVal = stack1;
}
```

B.3.2.6 main_MainMission_missionMemorySize

```
void main_MainMission_missionMemorySize(int32_t var1, int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 1000000;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.3.2.7 main_Barrier_isAlreadyTriggered

```
void main_Barrier_isAlreadyTriggered(int32_t var1, int32_t var2, int32_t * retVal) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack1 = ((main_Barrier *)  ((uintptr_t)stack1))->flag;
  stack2 = var2;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray5ID) {
    java_lang_BooleanArray5_load(stack1, stack2, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray4ID) {
    java_lang_BooleanArray4_load(stack1, stack2, &stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))->classID == java_lang_BooleanArray1ID) {
    java_lang_BooleanArray1_load(stack1, stack2, &stack1);
  } else if (((java_lang_Object*) ((uintptr_t)stack1))->classID == java_lang_BooleanArray3ID) {
    java_lang_BooleanArray3_load(stack1, stack2, &stack1);
  } else if (((java_lang_Object*) ((uintptr_t)stack1))->classID == java_lang_BooleanArray2ID) {
    java_lang_BooleanArray2_load(stack1, stack2, &stack1);
  }
  releaseLock(var1);
  *retVal = stack1;
}
```

B.3.2.8 main_MainSequence_init

```
void main_MainSequence_init(int32_t var1) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&stack4);
  if (((java_lang_Object*) ((uintptr_t)stack4))->classID == javax_safetycritical_PrioritySchedulerID) {
    javax_safetycritical_PriorityScheduler_getMaxPriority(stack4, &stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4);
  stack3 = newObject(javax_realtime_memory_ScopeParametersID);
  stack4 = stack3;
  stack5 = 0;
  stack6 = 702000;
  stack7 = 0;
  stack8 = 20000;
  stack9 = 0;
  stack10 = 100000;
  stack11 = 0;
  stack12 = 200000;
  javax_realtime_memory_ScopeParameters_init(stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12);
  stack4 = newObject(javax_realtime_ConfigurationParametersID);
  stack5 = stack4;
  stack6 = -1;
  stack7 = -1;
  stack8 = newObject(java_lang_LongArray1ID);
  stack9 = stack8;
  stack10 = 0;
  stack11 = 6144;
  java_lang_LongArray1_init(stack9, stack10, stack11);
  javax_realtime_ConfigurationParameters_init(stack5, stack6, stack7, stack8);
  javax_safetycritical_MissionSequencer_init(stack1, stack2, stack3, stack4);

}
```

B.3.2.9 main_MySafelet_getSequencer

```
void main_MySafelet_getSequencer(int32_t var1, int32_t * retVal) {
  int32_t stack1, stack2;
  stack1 = newObject(main_MainSequenceID);
  stack2 = stack1;
  main_MainSequence_init(stack2);
  *retVal = stack1;
}
```

B.3.2.10 main_MainMission_init

```
void main_MainMission_init(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  javax_safetycritical_Mission_init(stack1);

}
```

B.3.2.11 main_Barrier_init

```
void main_Barrier_init(int32_t var1, int32_t var2,
↪int32_t var3) {
  int32_t stack1, stack2;
  stack1 = var1;
  java_lang_Object_init(stack1);
  stack1 = var1;
  javax_safetycritical_PriorityScheduler_instance(&
  ↪stack2);
  if (((java_lang_Object*)  ((uintptr_t)stack2))->
  ↪classID == javax_safetycritical_PrioritySchedulerID)
  ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
    ↪(stack2, & stack2);
  }
  javax_safetycritical_Services_setCeiling(stack1,
  ↪stack2);
  stack1 = var1;
  stack2 = var2;
  java_lang_BooleanArray_newArray(stack2, & stack2);
  ((main_Barrier *) ((uintptr_t)stack1))->flag = stack2;
  stack1 = var1;
  stack2 = var3;
  ((main_Barrier *) ((uintptr_t)stack1))->e = stack2;

}
```

B.3.2.12 main_Button_handleAsyncEvent

```
void main_Button_handleAsyncEvent(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  stack1 = ((main_Button *)  ((uintptr_t)stack1))->event
  ↪;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_FireHandlerID) {
    javax_safetycritical_AperiodicEventHandler_release(
    ↪stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_LaunchHandlerID) {
    javax_safetycritical_AperiodicEventHandler_release(
    ↪stack1);
  }

}
```

B.3.2.13 main_Barrier_trigger

```
void main_Barrier_trigger(int32_t var1, int32_t var2) {
  int32_t stack1, stack2, stack3;
  stack1 = var1;
  stack1 = ((main_Barrier *)  ((uintptr_t)stack1))->flag
  ↪;
  stack2 = var2;
  stack3 = 1;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == java_lang_BooleanArray5ID) {
    java_lang_BooleanArray5_store(stack1, stack2, stack3
    ↪);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray4ID) {
    java_lang_BooleanArray4_store(stack1, stack2, stack3
    ↪);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == java_lang_BooleanArray1ID) {
    java_lang_BooleanArray1_store(stack1, stack2, stack3
    ↪);
  } else if (((java_lang_Object*) ((uintptr_t)stack1))->classID == java_lang_BooleanArray3ID) {
    java_lang_BooleanArray3_store(stack1, stack2, stack3);
  } else if (((java_lang_Object*) ((uintptr_t)stack1))->classID == java_lang_BooleanArray2ID) {
    java_lang_BooleanArray2_store(stack1, stack2, stack3);
  }
  stack1 = var1;
  if (((java_lang_Object*) ((uintptr_t)stack1))->classID == main_BarrierID) {
    takeLock(stack1);
    main_Barrier_isOkToFire(stack1, &stack1);
  }
  if (!(stack1 == 0)) {
    stack1 = var1;
    stack1 = ((main_Barrier *) ((uintptr_t)stack1))->e;
    if (((java_lang_Object*) ((uintptr_t)stack1))->classID == main_FireHandlerID) {
      javax_safetycritical_AperiodicEventHandler_release(stack1);
    } else if (((java_lang_Object*) ((uintptr_t)stack1))->classID == main_LaunchHandlerID) {
      javax_safetycritical_AperiodicEventHandler_release(stack1);
    }
    stack1 = var1;
    takeLock(stack1);
    main_Barrier_reset(stack1);
  }
  releaseLock(var1);
}
```

B.3.2.14 main_MainSequence_getNextMission

```
void main_MainSequence_getNextMission(int32_t var1, int32_t * retVal) {
  int32_t stack1, stack2;
  stack1 = newObject(main_MainMissionID);
  stack2 = stack1;
  main_MainMission_init(stack2);
  *retVal = stack1;
}
```

B.3.2.15 main_MySafelet_init

```
void main_MySafelet_init(int32_t var1) {
  int32_t stack1;
  stack1 = var1;
  java_lang_Object_init(stack1);

}
```

B.3.2.16 main_MySafelet_initializeApplication

```
void main_MySafelet_initializeApplication(int32_t var1) {

}
```

B.3.2.17 main_Button_init

```
void main_Button_init(int32_t var1, int32_t var2, int32_t var3, int32_t var4, int32_t var5, int32_t var6) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6, stack7, stack8, stack9, stack10, stack11, stack12, stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&stack4);
  if (((java_lang_Object*) ((uintptr_t)stack4))->classID == javax_safetycritical_PrioritySchedulerID) {
    javax_safetycritical_PriorityScheduler_getNormPriority
    ↪(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4)
  ↪;
  stack3 = newObject(javax_realtime_PeriodicParametersID
  ↪);
  stack4 = stack3;
  stack5 = newObject(javax_realtime_RelativeTimeID);
  stack6 = stack5;
  stack7 = var5;
  stack8 = var6;
  stack9 = 0;
  javax_realtime_RelativeTime_init(stack6, stack7,
  ↪stack8, stack9);
  stack6 = newObject(javax_realtime_RelativeTimeID);
  stack7 = stack6;
  stack8 = var3;
  stack9 = var4;
  stack10 = 0;
  javax_realtime_RelativeTime_init(stack7, stack8,
  ↪stack9, stack10);
  javax_realtime_PeriodicParameters_init(stack4, stack5,
  ↪ stack6);
  stack4 = newObject(
  ↪javax_realtime_memory_ScopeParametersID);
  stack5 = stack4;
  stack6 = 0;
  stack7 = 40000;
  stack8 = 0;
  stack9 = 20000;
  stack10 = 0;
  stack11 = 0;
  stack12 = 0;
  stack13 = 0;
  javax_realtime_memory_ScopeParameters_init(stack5,
  ↪stack6, stack7, stack8, stack9, stack10, stack11,
  ↪stack12, stack13);
  stack5 = newObject(
  ↪javax_realtime_ConfigurationParametersID);
  stack6 = stack5;
  stack7 = -1;
  stack8 = -1;
  stack9 = newObject(java_lang_LongArray1ID);
  stack10 = stack9;
  stack11 = 0;
  stack12 = 6144;
  java_lang_LongArray1_init(stack10, stack11, stack12);
  javax_realtime_ConfigurationParameters_init(stack6,
  ↪stack7, stack8, stack9);
  javax_safetycritical_PeriodicEventHandler_init(stack1,
  ↪ stack2, stack3, stack4, stack5);
  stack1 = var1;
  stack2 = var2;
  ((main_Button *) ((uintptr_t)stack1))->event = stack2;

}
```

B.3.2.18 main_FireHandler_init

```
void main_FireHandler_init(int32_t var1, int32_t var2,
↪int32_t var3) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6
  ↪, stack7, stack8, stack9, stack10, stack11, stack12,
  ↪ stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID
  ↪);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&
  ↪stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->
  ↪classID == javax_safetycritical_PrioritySchedulerID)
  ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
    ↪(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4)
  ↪;
  stack3 = newObject(
  ↪javax_realtime_AperiodicParametersID);
  stack4 = stack3;
  javax_realtime_AperiodicParameters_init(stack4);
  stack4 = newObject(
  ↪javax_realtime_memory_ScopeParametersID);
  stack5 = stack4;
  stack6 = 0;
  stack7 = 40000;
  stack8 = 0;
  stack9 = 20000;
  stack10 = 0;
  stack11 = 0;
  stack12 = 0;
  stack13 = 0;
  javax_realtime_memory_ScopeParameters_init(stack5,
  ↪stack6, stack7, stack8, stack9, stack10, stack11,
  ↪stack12, stack13);
  stack5 = newObject(
  ↪javax_realtime_ConfigurationParametersID);
  stack6 = stack5;
  stack7 = -1;
  stack8 = -1;
  stack9 = newObject(java_lang_LongArray1ID);
  stack10 = stack9;
  stack11 = 0;
  stack12 = 6144;
  java_lang_LongArray1_init(stack10, stack11, stack12);
  javax_realtime_ConfigurationParameters_init(stack6,
  ↪stack7, stack8, stack9);
  javax_safetycritical_AperiodicEventHandler_init(stack1
  ↪, stack2, stack3, stack4, stack5);
  stack1 = var1;
  stack2 = var2;
  ((main_FireHandler *) ((uintptr_t)stack1))->barrier =
  ↪stack2;
  stack1 = var1;
  stack2 = var3;
  ((main_FireHandler *) ((uintptr_t)stack1))->id =
  ↪stack2;

}
```

B.3.2.19 main_MySafelet_immortalMemorySize

```
void main_MySafelet_immortalMemorySize(int32_t var1,
↪int32_t * retVal_msb, int32_t * retVal_lsb) {
  int32_t stack1, stack2;
  stack1 = 0;
  stack2 = 0;
  *retVal_lsb = stack2;
  *retVal_msb = stack1;
}
```

B.3.2.20 main_Barrier_reset

```
void main_Barrier_reset(int32_t var1) {
  int32_t var2;
  int32_t stack1, stack2, stack3;
  stack1 = 0;
  var2 = stack1;
  stack1 = var2;
  stack2 = var1;
  stack2 = ((main_Barrier *) ((uintptr_t)stack2))->flag
  ↪;
  if (((java_lang_Object*) ((uintptr_t)stack2))->
  ↪classID == java_lang_BooleanArray5ID) {
    java_lang_BooleanArray5_length(stack2, & stack2);
  } else if (((java_lang_Object*) ((uintptr_t)stack2))
  ↪->classID == java_lang_BooleanArray4ID) {
    java_lang_BooleanArray4_length(stack2, & stack2);
  } else if (((java_lang_Object*) ((uintptr_t)stack2))
  ↪->classID == java_lang_BooleanArray1ID) {
    java_lang_BooleanArray1_length(stack2, & stack2);
  } else if (((java_lang_Object*) ((uintptr_t)stack2))
  ↪->classID == java_lang_BooleanArray3ID) {
    java_lang_BooleanArray3_length(stack2, & stack2);
  } else if (((java_lang_Object*) ((uintptr_t)stack2))
  ↪->classID == java_lang_BooleanArray2ID) {
    java_lang_BooleanArray2_length(stack2, & stack2);
  }
  while (stack1 < stack2) {
    stack1 = var1;
    stack1 = ((main_Barrier *)  ((uintptr_t)stack1))->
    ↪flag;
    stack2 = var2;
    stack3 = 0;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->
    ↪classID == java_lang_BooleanArray5ID) {
      java_lang_BooleanArray5_store(stack1, stack2,
      ↪stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_BooleanArray4ID) {
      java_lang_BooleanArray4_store(stack1, stack2,
      ↪stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_BooleanArray1ID) {
      java_lang_BooleanArray1_store(stack1, stack2,
      ↪stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_BooleanArray3ID) {
      java_lang_BooleanArray3_store(stack1, stack2,
      ↪stack3);
    } else if (((java_lang_Object*)  ((uintptr_t)stack1)
    ↪)->classID == java_lang_BooleanArray2ID) {
      java_lang_BooleanArray2_store(stack1, stack2,
      ↪stack3);
    }
    var2 = var2 + 1;
    stack1 = var2;
    stack2 = var1;
    stack2 = ((main_Barrier *)  ((uintptr_t)stack2))->
    ↪flag;
    if (((java_lang_Object*)  ((uintptr_t)stack2))->
    ↪classID == java_lang_BooleanArray5ID) {
      java_lang_BooleanArray5_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2)
    ↪)->classID == java_lang_BooleanArray4ID) {
      java_lang_BooleanArray4_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2)
    ↪)->classID == java_lang_BooleanArray1ID) {
      java_lang_BooleanArray1_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2)
    ↪)->classID == java_lang_BooleanArray3ID) {
      java_lang_BooleanArray3_length(stack2, &stack2);
    } else if (((java_lang_Object*)  ((uintptr_t)stack2)
    ↪)->classID == java_lang_BooleanArray2ID) {
      java_lang_BooleanArray2_length(stack2, &stack2);
    }

  }
  releaseLock(var1);
}
```

B.3.2.21 main_MainMission_initialize

```
void main_MainMission_initialize(int32_t var1) {
  int32_t var2, var3, var4, var5;
  int32_t stack1, stack2, stack3, stack4, stack5, stack6
  ↪, stack7;
  stack1 = newObject(main_LaunchHandlerID);
  stack2 = stack1;
  main_LaunchHandler_init(stack2);
  var2 = stack1;
  stack1 = var2;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_FireHandlerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_LaunchHandlerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }
  stack1 = newObject(main_BarrierID);
  stack2 = stack1;
  stack3 = 2;
  stack4 = var2;
  main_Barrier_init(stack2, stack3, stack4);
  var3 = stack1;
  stack1 = newObject(main_FireHandlerID);
  stack2 = stack1;
  stack3 = var3;
  stack4 = 0;
  main_FireHandler_init(stack2, stack3, stack4);
  var4 = stack1;
  stack1 = newObject(main_FireHandlerID);
  stack2 = stack1;
  stack3 = var3;
  stack4 = 1;
  main_FireHandler_init(stack2, stack3, stack4);
  var5 = stack1;
  stack1 = var4;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_FireHandlerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_LaunchHandlerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }
  stack1 = var5;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_FireHandlerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  } else if (((java_lang_Object*)  ((uintptr_t)stack1))
  ↪->classID == main_LaunchHandlerID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }
  stack1 = newObject(main_ButtonID);
  stack2 = stack1;
  stack3 = var4;
  stack4 = 0;
  stack5 = 2000;
  stack6 = 0;
  stack7 = 0;
  main_Button_init(stack2, stack3, stack4, stack5,
  ↪stack6, stack7);
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_ButtonID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }
  stack1 = newObject(main_ButtonID);
  stack2 = stack1;
  stack3 = var5;
  stack4 = 0;
  stack5 = 9000;
  stack6 = 0;
  stack7 = 9000;
  main_Button_init(stack2, stack3, stack4, stack5,
  ↪stack6, stack7);
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_ButtonID) {
    javax_safetycritical_ManagedEventHandler_register(
    ↪stack1);
  }

}
```

B.3.2.22 main_LaunchHandler_init

```
void main_LaunchHandler_init(int32_t var1) {
  int32_t stack1, stack2, stack3, stack4, stack5, stack6
  ↪, stack7, stack8, stack9, stack10, stack11, stack12,
  ↪ stack13;
  stack1 = var1;
  stack2 = newObject(javax_realtime_PriorityParametersID
  ↪);
  stack3 = stack2;
  javax_safetycritical_PriorityScheduler_instance(&
  ↪stack4);
  if (((java_lang_Object*)  ((uintptr_t)stack4))->
  ↪classID == javax_safetycritical_PrioritySchedulerID)
  ↪ {
    javax_safetycritical_PriorityScheduler_getMaxPriority
    ↪(stack4, & stack4);
  }
  javax_realtime_PriorityParameters_init(stack3, stack4)
  ↪;
  stack3 = newObject(
  ↪javax_realtime_AperiodicParametersID);
  stack4 = stack3;
  javax_realtime_AperiodicParameters_init(stack4);
  stack4 = newObject(
  ↪javax_realtime_memory_ScopeParametersID);
  stack5 = stack4;
  stack6 = 0;
  stack7 = 40000;
  stack8 = 0;
  stack9 = 20000;
  stack10 = 0;
  stack11 = 0;
  stack12 = 0;
  stack13 = 0;
  javax_realtime_memory_ScopeParameters_init(stack5,
  ↪stack6, stack7, stack8, stack9, stack10, stack11,
  ↪stack12, stack13);
  stack5 = newObject(
  ↪javax_realtime_ConfigurationParametersID);
  stack6 = stack5;
  stack7 = -1;
  stack8 = -1;
  stack9 = newObject(java_lang_LongArray1ID);
  stack10 = stack9;
  stack11 = 0;
  stack12 = 6144;
  java_lang_LongArray1_init(stack10, stack11, stack12);
  javax_realtime_ConfigurationParameters_init(stack6,
  ↪stack7, stack8, stack9);
  javax_safetycritical_AperiodicEventHandler_init(stack1
  ↪, stack2, stack3, stack4, stack5);

}
```

B.3.2.23 main_FireHandler_handleAsyncEvent

```
void main_FireHandler_handleAsyncEvent(int32_t var1) {
  int32_t stack1, stack2;
  stack1 = var1;
  stack1 = ((main_FireHandler *)  ((uintptr_t)stack1))->
  ↪id;
  devices_Console_write(stack1);
  stack1 = var1;
  stack1 = ((main_FireHandler *)  ((uintptr_t)stack1))->
  ↪barrier;
  stack2 = var1;
  stack2 = ((main_FireHandler *)  ((uintptr_t)stack2))->
  ↪id;
  if (((java_lang_Object*)  ((uintptr_t)stack1))->
  ↪classID == main_BarrierID) {
    takeLock(stack1);
    main_Barrier_isAlreadyTriggered(stack1, stack2, &
    ↪stack1);
  }
  if (stack1 == 0) {
    stack1 = var1;
    stack1 = ((main_FireHandler *)  ((uintptr_t)stack1))
    ↪->barrier;
    stack2 = var1;
    stack2 = ((main_FireHandler *)  ((uintptr_t)stack2))
    ↪->id;
    if (((java_lang_Object*)  ((uintptr_t)stack1))->
    ↪classID == main_BarrierID) {
      takeLock(stack1);
      main_Barrier_trigger(stack1, stack2);
    }

  } else {

  }
}
```

Bibliography

[1] aicas GmbH. *JamaicaVM 8.1 — User Manual.* 2017. URL: https://www.aicas.com/cms/sites/default/files/JamaicaVM-8.1-Manual-A4_0.pdf.

[2] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. "Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* OOPSLA '01. Tampa Bay, FL, USA: ACM, 2001, pp. 108–124. ISBN: 1-58113-335-9. DOI: 10.1145/504282.504291. URL: http://doi.acm.org/10.1145/504282.504291.

[3] Jeppe L. Andersen, Mikkel Todberg, Andreas E. Dalsgaard, and René Rydhof Hansen. "Worst-case Memory Consumption Analysis for SCJ". In: *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems.* JTRES '13. New York, NY, USA: ACM, 2013, pp. 2–10. ISBN: 978-1-4503-2166-2. DOI: 10.1145/2512989.2513000. URL: http://doi.acm.org/10.1145/2512989.2513000.

[4] Austin Armbruster et al. "A Real-time Java Virtual Machine with Applications in Avionics". In: *ACM Trans. Embed. Comput. Syst.* 7.1 (Dec. 2007), 5:1–5:49. ISSN: 1539-9087. DOI: 10.1145/1324969.1324974. URL: http://doi.acm.org/10.1145/1324969.1324974.

[5] Jacques J. Arsac. "Syntactic Source to Source Transforms and Program Manipulation". In: *Commun. ACM* 22.1 (Jan. 1979), pp. 43–54. ISSN: 0001-0782. DOI: 10.1145/359046.359057. URL: http://doi.acm.org/10.1145/359046.359057.

[6] Atego. *Atego Perc Pico - Products - Atego.* 2015. URL: http://www.atego.com/products/atego-perc-pico/.

[7] RJR Back. "On correct refinement of programs". In: *Journal of Computer and System Sciences* 23.1 (1981), pp. 49–68.

[8] Patrick Bahr and Graham Hutton. "Calculating correct compilers". In: *Journal of Functional Programming* 25 (2015). ISSN: 1469-7653. DOI: 10.1017/S0956796815000180. URL: http://journals.cambridge.org/article_S0956796815000180.

[9] J. Baker et al. "A Real-time Java Virtual Machine for Avionics - An Experience Report". In: *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE.* Apr. 2006, pp. 384–396. DOI: 10.1109/RTAS.2006.7.

[10] Robert Balzer, Neil Goldman, and David Wile. "On the Transformational Implementation Approach to Programming". In: *Proceedings of the 2Nd International Conference on Software Engineering.* ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 337–344. URL: http://dl.acm.org/citation.cfm?id=800253.807696.

[11] F. L. Bauer. "Programming As an Evolutionary Process". In: *Proceedings of the 2nd International Conference on Software Engineering.* ICSE '76. San Francisco, California,

USA: IEEE Computer Society Press, 1976, pp. 223–234. URL: http://dl.acm.org/citation.cfm?id=800253.807679.

[12] James Baxter. *Requirements for Safety-Critical Java Virtual Machines.* Technical report. University of York, 2015. URL: http://www.cs.york.ac.uk/circus/publications/techreports/reports/scjvm-requirements.pdf.

[13] James Baxter. *An Approach to Verification of Ahead-of-time Compilation for Safety-Critical Java (extended version).* University of York, 2018. URL: https://www.cs.york.ac.uk/circus/publications/techreports/reports/18-baxter-extended.pdf.

[14] James Baxter, Ana Cavalcanti, Andy Wellings, and Leo Freitas. "Safety-Critical Java Virtual Machine Services". In: *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems.* JTRES '15. New York, NY, USA: ACM, 2015, 7:1–7:10. ISBN: 978-1-4503-3644-4. DOI: 10.1145/2822304.2822307.

[15] Lucila M.S. Bento et al. "Dijkstra graphs". In: *Discrete Applied Mathematics* (2017). ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2017.07.033.

[16] Peter Bertelsen. "Dynamic semantics of Java bytecode". In: *Future Generation Computer Systems* 16.7 (2000), pp. 841–850.

[17] T.F. Bissyande et al. "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects". In: *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual.* July 2013, pp. 303–312. DOI: 10.1109/COMPSAC.2013.55.

[18] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal Verification of a C Compiler Front-End". English. In: *FM 2006: Formal Methods.* Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 460–475. ISBN: 978-3-540-37215-8. DOI: 10.1007/11813040_31. URL: http://dx.doi.org/10.1007/11813040_31.

[19] Paulo Borba and Augusto Sampaio. "Basic Laws of ROOL: an object-oriented language". In: *Revista de Informática Teórica e Aplicada* 7.1 (2000), pp. 49–68.

[20] Rodney M Burstall and Peter J Landin. "Programs and their proofs: an algebraic approach". In: *Machine Intelligence 4.* Ed. by Bernard Meltzer and Donald Michie. Edinburgh University Press, 1969, pp. 17–44.

[21] Bettina Buth et al. "Provably correct compiler development and implementation". English. In: *Compiler Construction.* Ed. by Uwe Kastens and Peter Pfahler. Vol. 641. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 141–155. ISBN: 978-3-540-55984-9. DOI: 10.1007/3-540-55984-1_14. URL: http://dx.doi.org/10.1007/3-540-55984-1_14.

[22] James Caska and Martin Schoeberl. "Java Dust: How Small Can Embedded Java Be?" In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* Ed. by Andy Wellings and Anders P. Ravn. ACM, 2011.

[23] Ana Cavalcanti, Andy Wellings, and Jim Woodcock. "The Safety-Critical Java Memory Model: A Formal Account". English. In: *FM 2011: Formal Methods.* Ed. by Michael Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 246–261. ISBN: 978-3-642-21436-3. DOI: 10.1007/978-3-642-21437-0_20. URL: http://dx.doi.org/10.1007/978-3-642-21437-0_20.

[24] Ana Cavalcanti and Jim Woodcock. "ZRC – A Refinement Calculus for Z". In: *Formal Aspects of Computing* 10.3 (Mar. 1998), pp. 267–289. ISSN: 1433-299X. DOI: 10.1007/s001650050016. URL: https://doi.org/10.1007/s001650050016.

[25] Ana Cavalcanti et al. "Safety-critical Java in Circus". In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* Ed. by Andy Wellings and Anders P. Ravn. ACM, 2011, pp. 20–29.

[26] Ana Cavalcanti et al. "Safety-critical Java programs from Circus models". In: *Real-Time Systems* 49.5 (2013), pp. 614–667.

[27] Zhiqun Chen. *Java card technology for smart cards: architecture and programmer's guide.* Addison-Wesley Professional, 2000.

[28] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. "MultiJava: Design Rationale, Compiler Implementation, and Applications". In: *ACM Transactions on Programming Languages and Systems* 28.3 (2006), pp. 517–575.

[29] A. Coglio, A. Goldberg, and Zhenyu Qian. "Toward a provably-correct implementation of the JVM bytecode verifier". In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings.* Vol. 2. 2000, 403–410 vol.2. DOI: `10.1109/DISCEX.2000.821537`.

[30] Angelo Corsaro and Douglas C. Schmidt. "The Design and Performance of the jRate Real-Time Java Implementation". In: *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE.* Ed. by Robert Meersman and Zahir Tari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 900–921. ISBN: 978-3-540-36124-4.

[31] Robert I Davis and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems". In: *ACM Computing Surveys (CSUR)* 43.4 (2011), p. 35.

[32] Edsger W. Dijkstra. "Structured Programming". In: ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK, UK: Academic Press Ltd., 1972. Chap. Notes on Structured Programming, pp. 1–82. ISBN: 0-12-200550-3. URL: `http://dl.acm.org/citation.cfm?id=1243380.1243381`.

[33] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Communications of the ACM* 18.8 (1975), pp. 453–457. ISSN: 0001-0782.

[34] Adolfo Duran. "An Algebraic Approach to the Design of Compilers for Object-Oriented Languages". PhD Thesis. Universidade Federalde Pernambuco, 2005.

[35] Adolfo Duran, Ana Cavalcanti, and Augusto Sampaio. "An algebraic approach to the design of compilers for object-oriented languages". In: *Formal aspects of computing* 22.5 (2010), pp. 489–535.

[36] École Polytechnique Fédérale de Lausanne (EPFL). *The Scala Programming Language.* 2015. URL: `http://scala-lang.org/` (visited on 03/09/2015).

[37] Simon Foster, Frank Zeyda, and Jim Woodcock. "Isabelle/UTP: A Mechanised Theory Engineering Framework". English. In: *Unifying Theories of Programming.* Ed. by David Naumann. Vol. 8963. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 21–41. ISBN: 978-3-319-14805-2. DOI: `10.1007/978-3-319-14806-9_2`. URL: `http://dx.doi.org/10.1007/978-3-319-14806-9_2`.

[38] Leo Freitas, James Baxter, Ana Cavalcanti, and Andy Wellings. "Modelling and Verifying a Priority Scheduler for an SCJ Runtime Environment". In: *Integrated Formal Methods: 12th International Conference, iFM 2016.* Ed. by Erika Ábrahám and Marieke Huisman. Cham: Springer International Publishing, 2016, pp. 63–78. ISBN: 978-3-319-33693-0. DOI: `10.1007/978-3-319-33693-0_5`.

[39] Joseph Goguen et al. "An introduction to OBJ 3". In: *Conditional Term Rewriting Systems.* Ed. by S. Kaplan and J. P. Jouannaud. Springer. 1988, pp. 258–263.

[40] James Gosling and Greg Bollella. *The Real-Time Specification for Java.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201703238.

[41] James Gosling et al. *The Java Language Specification.* Addison-Wesley, 2013.

[42] Florian Haftmann and Tobias Nipkow. “A code generator framework for Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science* 4732 (2007), pp. 128–143.

[43] Pieter H. Hartel and Luc Moreau. “Formalizing the Safety of Java, the Java Virtual Machine, and Java Card”. In: *ACM Computing Surveys* 33.4 (Dec. 2001), pp. 517–558. ISSN: 0360-0300. DOI: `10.1145/503112.503115`.

[44] CAR Hoare. “Refinement algebra proves correctness of compiling specifications”. In: *3rd Refinement Workshop*. Ed. by Carroll Morgan and Jim Woodcock. 1991, pp. 33–48. ISBN: 3540196242.

[45] CAR Hoare and FK Hanna. “Programs are predicates [and discussion]”. In: *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 312.1522 (1984), pp. 475–489.

[46] CAR Hoare, He Jifeng, and Augusto Sampaio. “Normal form approach to compiler design”. In: *Acta informatica* 30.8 (1993), pp. 701–739.

[47] Charles Antony Richard Hoare and He Jifeng. *Unifying theories of programming.* Prentice Hall, 1998.

[48] Charles Antony Richard Hoare et al. “Laws of programming”. In: *Communications of the ACM* 30.8 (1987), pp. 672–686.

[49] Cay S. Horstmann and Gary Cornell. *Core Java 2: Volume I, Fundamentals, Sixth Edition.* 6th. Pearson Education, 2002. ISBN: 0130471771.

[50] Mark Jones. “The functions of Java bytecode”. In: *Workshop on the Formal Underpinnings of the Java Paradigm.* 1998.

[51] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall international series in computer science. Prentice Hall, 1993. ISBN: 9780130202499.

[52] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. “Exhaustive testing of safety critical Java”. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.* ACM. 2010, pp. 164–174.

[53] Gerwin Klein and Tobias Nipkow. “Verified bytecode verifiers”. In: *Theoretical Computer Science* 298.3 (2003), pp. 583–626.

[54] Gerwin Klein and Tobias Nipkow. “A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler”. In: *ACM Transactions on Programming Languages and Systems* 28.4 (2006), pp. 619–695. ISSN: 0164-0925. DOI: `10.1145/1146809.1146811`.

[55] Donald E Knuth and Luis Trabb Pardo. “The early development of programming languages”. In: *A history of computing in the twentieth century* (1980), pp. 197–273.

[56] Andrew Kornecki and Janusz Zalewski. “Certification of software for real-time safety-critical systems: state of the art”. English. In: *Innovations in Systems and Software Engineering* 5.2 (2009), pp. 149–161. ISSN: 1614-5046. DOI: `10.1007/s11334-009-0088-1`. URL: `http://dx.doi.org/10.1007/s11334-009-0088-1`.

[57] D. Leinenbach, W. Paul, and E. Petrova. “Towards the formal verification of a C0 compiler: code generation and implementation correctness”. In: *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on.* Sept. 2005, pp. 2–11. DOI: `10.1109/SEFM.2005.51`.

[58] Xavier Leroy. “Java Bytecode Verification: Algorithms and Formalizations”. In: *Journal of Automated Reasoning* 30.3 (2003), pp. 235–269. ISSN: 1573-0670. DOI: `10.1023/A:1025055424017`. URL: `http://dx.doi.org/10.1023/A:1025055424017`.

[59] Xavier Leroy. “A formally verified compiler back-end”. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446.

[60] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[61] Xavier Leroy. *The CompCert C verified compiler.* 2012.

[62] Pierre Letouzey. “A new extraction for Coq”. In: *Types for proofs and program.* Ed. by Herman Geuvers and Freek Wiedijk. Springer, 2003, pp. 200–219.

[63] Pierre Letouzey. “Extraction in coq: An overview”. In: *Logic and Theory of Algorithms.* Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Springer, 2008, pp. 359–369.

[64] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification.* Pearson Education, 2014.

[65] Andreas Lochbihler. “Verifying a compiler for Java threads”. In: *Programming languages and systems.* Ed. by Andrew D. Gordon. Springer, 2010, pp. 427–447.

[66] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler.* KIT Scientific Publishing, 2012.

[67] D. Locke et al. *Safety-Critical Java Technology Specification.* Draft. Version 0.94. The Open Group, June 25, 2013. URL: `https://jcp.org/aboutJava/communityprocess/edr/jsr302/index2.html` (visited on 11/18/2014).

[68] Matthew Luckcuck. “Safety-Critical Java Level 2: Applications, Modelling, and Verification”. PhD thesis. University of York, 2016. URL: `http://etheses.whiterose.ac.uk/id/eprint/17307`.

[69] Kasper Søe Luckow, Bent Thomsen, and Stephan Erbs Korsholm. “HVMTP: A Time Predictable and Portable Java Virtual Machine for Hard Real-Time Embedded Systems”. In: *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems.* Ed. by Wolfgang Puffitsch. JTRES ’14. Niagara Falls, NY, USA: ACM, 2014, 107:107–107:116. ISBN: 978-1-4503-2813-5. DOI: `10.1145/2661020.2661022`. URL: `http://doi.acm.org/10.1145/2661020.2661022`.

[70] Petra Malik. “A retrospective on CZT”. In: *Software: Practice and Experience* 41.2 (2011), pp. 179–188. ISSN: 1097-024X. DOI: `10.1002/spe.1015`. URL: `http://dx.doi.org/10.1002/spe.1015`.

[71] Petra Malik and Mark Utting. “CZT: A Framework for Z Tools”. English. In: *ZB 2005: Formal Specification and Development in Z and B.* Ed. by Helen Treharne, Steve King, Martin Henson, and Steve Schneider. Vol. 3455. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 65–84. ISBN: 978-3-540-25559-8. DOI: `10.1007/11415787_5`. URL: `http://dx.doi.org/10.1007/11415787_5`.

[72] Chris Marriott. “Checking Memory Safety of Level 1 Safety-Critical Java Programs using Static-Analysis without Annotations”. PhD thesis. University of York, 2014.

[73] The Coq development team. *The Coq proof assistant reference manual.* Version 8.0. LogiCal Project. 2004. URL: `http://coq.inria.fr`.

[74] James Mc Enery, David Hickey, and Menouer Boubekeur. “Empirical Evaluation of Two Main-stream RTSJ Implementations”. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems.* JTRES ’07. Vienna, Austria: ACM, 2007, pp. 47–54. ISBN: 978-1-59593-813-8. DOI: `10.1145/1288940.1288947`. URL: `http://doi.acm.org/10.1145/1288940.1288947`.

[75] John McCarthy and James Painter. “Correctness of a compiler for arithmetic expressions”. In: *Mathematical aspects of computer science* 1 (1967).

[76] Robin Milner. “Implementation and Applications of Scott’s Logic for Computable Functions”. In: *SIGPLAN Not.* 7.1 (Jan. 1972), pp. 1–6. ISSN: 0362-1340. DOI: 10.1145/942578.807067. URL: http://doi.acm.org/10.1145/942578.807067.

[77] Robin Milner and Richard Weyhrauch. “Proving compiler correctness in a mechanized logic”. In: *Machine Intelligence* 7 (1972), pp. 51–70.

[78] Alvaro Heiji Miyazawa. “Formal verification of implementations of Stateflow charts”. PhD thesis. University of York, 2012.

[79] Carroll Morgan. *Programming from specifications.* Prentice-Hall, Inc., 1990.

[80] F Lockwood Morris. “Advice on structuring compilers and proving them correct”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM. 1973, pp. 144–152.

[81] Joseph M Morris. “A theoretical basis for stepwise refinement and the programming calculus”. In: *Science of Computer programming* 9.3 (1987), pp. 287–306.

[82] Motor Industry Software Reliability Association Guidelines. *Guidelines for Use of the C Language in Critical Systems.* 2012.

[83] D. Mulchandani. “Java for embedded systems”. In: *Internet Computing, IEEE* 2.3 (May 1998), pp. 30–39. ISSN: 1089-7801. DOI: 10.1109/4236.683797.

[84] Kelvin Nilsen. “Harmonizing Alternative Approaches to Safety-critical Development with Java”. In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems.* Ed. by Andy Wellings and Anders P. Ravn. JTRES ’11. York, United Kingdom: ACM, 2011, pp. 54–63. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043920. URL: http://doi.acm.org/10.1145/2043910.2043920.

[85] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. “μJava: Embedding a Programming Language in a Theorem Prover”. In: *Foundations of Secure Computation, volume 175 of NATO Science Series F: Computer and Systems Sciences.* Ed. by Friedrich L Bauer and Ralf Steinbrüggen. IOS Press, 2000, pp. 117–144.

[86] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* Vol. 2283. Springer Science & Business Media, 2002.

[87] Martin Odersky and Philip Wadler. “Pizza into Java: Translating Theory into Practice”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Ed. by Peter Lee, Fritz Henglein, and Neil D. Jones. ACM, 1997, pp. 146–159.

[88] M. V. M. Oliveira. “Formal Derivation of State-Rich Reactive Programs using *Circus*”. PhD thesis. Department of Computer Science - University of York, 2006. URL: https://www.cs.york.ac.uk/circus/publications/techreports/reports/06-oliveira.pdf.

[89] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. “A UTP semantics for Circus”. English. In: *Formal Aspects of Computing* 21.1-2 (2009), pp. 3–32. ISSN: 0934-5043. DOI: 10.1007/s00165-007-0052-5. URL: http://dx.doi.org/10.1007/s00165-007-0052-5.

[90] Oracle Corporation. *Java Platform, Micro Edition (Java ME).* 2014. URL: http://www.oracle.com/technetwork/java/embedded/javame/index.html (visited on 11/25/2014).

[91] Juan Perna, Jim Woodcock, Augusto Sampaio, and Juliano Iyoda. “Correct hardware synthesis. An algebraic approach”. In: *Acta informatica* 48.7-8 (2011), pp. 363–396.

[92] Juan Ignacio Perna. “A verified compiler for Handel-C”. PhD Thesis. University of York, 2010. URL: http://etheses.whiterose.ac.uk/585/.

[93] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. “Real Time Java on Resource-constrained Platforms with Fiji VM”. In: *Proceedings of the 7th International Workshop on Java*

Technologies for Real-Time and Embedded Systems. JTRES '09. Madrid, Spain: ACM, 2009, pp. 110–119. ISBN: 978-1-60558-732-5. DOI: 10.1145/1620405.1620421. URL: http://doi.acm.org/10.1145/1620405.1620421.

[94] Ales Plsek et al. "Developing Safety Critical Java Applications with oSCJ/L0". In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '10. Prague, Czech Republic: ACM, 2010, pp. 95–101. ISBN: 978-1-4503-0122-0. DOI: 10.1145/1850771.1850786. URL: http://doi.acm.org/10.1145/1850771.1850786.

[95] Wolfgang Polak. *Compiler Specification and Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1981. ISBN: 0387108866.

[96] Red Hat, Inc. *Ceylon: Welcome to Ceylon*. 2015. URL: http://ceylon-lang.org/ (visited on 03/09/2015).

[97] M Richard-Foy et al. "Use of PERC Pico for Safety Critical Java". In: *Conference Proceedings: Embedded Real-Time Software and Systems, Toulouse, France*. 2010.

[98] Juan Ricardo Rios Rivas and Martin Schoeberl. "Safety-Critical Java for Embedded Systems". PhD thesis. Technical University of Denmark (DTU), 2014.

[99] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[100] Mark Saaltink. "The Z/EVES system". English. In: *ZUM '97: The Z Formal Specification Notation*. Ed. by JonathanP. Bowen, MichaelG. Hinchey, and David Till. Vol. 1212. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 72–85. ISBN: 978-3-540-62717-3. DOI: 10.1007/BFb0027284. URL: http://dx.doi.org/10.1007/BFb0027284.

[101] Augusto Sampaio. "An algebraic approach to compiler design". PhD Thesis. Oxford University Computing Laboratory, 1993. ISBN: 0902928872.

[102] Augusto Sampaio. *An algebraic approach to compiler design*. World Scientific, 1997. ISBN: 9789810223915.

[103] Puntitra Sawadpong, Edward B Allen, and Byron J Williams. "Exception handling defects: An empirical study". In: *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*. IEEE. 2012, pp. 90–97.

[104] Martin Schoeberl. "Real-time garbage collection for Java". In: *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. Apr. 2006, 9 pp.–. DOI: 10.1109/ISORC.2006.66.

[105] Martin Schoeberl. "A Java processor architecture for embedded real-time systems". In: *Journal of Systems Architecture* 54.1 (2008), pp. 265 –286. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2007.06.001.

[106] Martin Schoeberl. "Memory Management for Safety-critical Java". In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '11. York, United Kingdom: ACM, 2011, pp. 47–53. ISBN: 978-1-4503-0731-4. DOI: 10.1145/2043910.2043919. URL: http://doi.acm.org/10.1145/2043910.2043919.

[107] Martin Schoeberl and Wolfgang Puffitsch. "Nonblocking Real-time Garbage Collection". In: *ACM Trans. Embed. Comput. Syst.* 10.1 (Aug. 2010), 6:1–6:28. ISSN: 1539-9087. DOI: 10.1145/1814539.1814545. URL: http://doi.acm.org/10.1145/1814539.1814545.

[108] Martin Schoeberl et al. "Certifiable Java for Embedded Systems". In: *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*. Ed. by Wolfgang Puffitsch. JTRES '14. Niagara Falls, NY, USA: ACM, 2014,

10:10–10:19. ISBN: 978-1-4503-2813-5. DOI: 10.1145/2661020.2661025. URL: http://doi.acm.org/10.1145/2661020.2661025.

[109] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. "Compiling java for low-end embedded systems". In: *ACM SIGPLAN Notices*. Ed. by Frank Mueller and Uli Kremer. Vol. 38. 7. ACM. 2003, pp. 42–50.

[110] Hans Søndergaard, Stephan E. Korsholm, and Anders P. Ravn. "Safety-critical Java for Low-end Embedded Platforms". In: *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. Ed. by Martin Schoeberl and Andy Wellings. JTRES '12. ACM, 2012, pp. 44–53. ISBN: 978-1-4503-1688-0. DOI: 10.1145/2388936.2388945.

[111] Thomas A. Standish, Dennis F. Kibler, and James M. Neighbors. "Improving and Refining Programs by Program Manipulation". In: *Proceedings of the 1976 Annual Conference*. ACM '76. Houston, Texas, USA: ACM, 1976, pp. 509–516. DOI: 10.1145/800191.805652. URL: http://doi.acm.org/10.1145/800191.805652.

[112] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine. Definition, Verification, Validation*. Springer-Verlag, 2001. ISBN: 3-540-42088-6.

[113] Martin Strecker. "Formal verification of a Java compiler in Isabelle". In: *Automated DeductionCADE-18*. Ed. by Andrei Voronkov. Springer, 2002, pp. 63–77.

[114] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. "More on Advice on Structuring Compilers and Proving Them Correct". In: *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*. Ed. by Hermann A. Maurer. London, UK: Springer-Verlag, 1979, pp. 596–615. ISBN: 3-540-09510-1.

[115] Isabella Thomm, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat. "KESO: An Open-source multi-JVM for Deeply Embedded Systems". In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. Ed. by Tomas Kalibera and Jan Vitek. ACM, 2010, pp. 109–119.

[116] Ankush Varma and Shuvra S Bhattacharyya. "Java-through-C compilation: An enabling technology for java in embedded systems". In: *Proceedings of the conference on Design, automation and test in Europe-Volume 3*. IEEE Computer Society. 2004, p. 30161.

[117] Malcolm Wallace and Colin Runciman. "An incremental garbage collector for embedded real-time systems". In: *Proceedings of the Chalmers Winter Meeting*. 1993, pp. 273–288.

[118] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

[119] Frank Zeyda, Ana Cavalcanti, and Andy Wellings. "The Safety-Critical Java Mission Model: A Formal Account". English. In: *Formal Methods and Software Engineering*. Ed. by Shengchao Qin and Zongyan Qiu. Vol. 6991. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 49–65. ISBN: 978-3-642-24558-9. DOI: 10.1007/978-3-642-24559-6_6. URL: http://dx.doi.org/10.1007/978-3-642-24559-6_6.

[120] Shuai Zhao, Andy Wellings, and Stephan Erbs Korsholm. "Supporting Multiprocessors in the Icecap Safety-Critical Java Run-Time Environment". In: *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '15. New York, NY, USA: ACM, 2015, 1:1–1:10. ISBN: 978-1-4503-3644-4. DOI: 10.1145/2822304.2822305. URL: http://doi.acm.org/10.1145/2822304.2822305.