Checking Memory Safety of Level 1 Safety-Critical Java Programs using Static-Analysis without Annotations

Christopher Alexander Marriott

PhD

University of York

Computer Science

September 2014

Abstract

Safety-Critical Java (SCJ) has been designed specifically to bring performance and reliability to the development of safety-critical Java programs. SCJ introduces a novel programming paradigm based on missions and handlers, and has been designed to ease certification. One of the distinguishing features of SCJ is its memory model, which is defined as a hierarchical structure of scoped-based memory areas. Unlike in Java programs, memory management is an important concern under the control of the programmer in SCJ; it is not sufficient to write a program that conforms to the specification as memory safety may still be broken.

By using static analysis techniques, it is possible to identify errors in programs before they are executed. Analysing at the source-code level allows for a precise analysis that abstracts away from machine details and unnecessary program details. As the SCJ paradigm is different to that of Java, it is not possible to apply existing tools and techniques for Java programs to SCJ.

This thesis describes a new static-checking technique for a comprehensive subset of SCJ programs (comparable to Ravenscar Ada) that automatically checks for memory-safety violations at the source-code level without the need for user-added annotations. An abstract language (*SCJ-mSafe*) is used to describe the aspects of SCJ programs required to check memory safety, and a set of inference rules define what it means for each aspect to be memory safe. By using a points-to environment and automatically-generated method properties, it is possible to produce a model of the execution of an SCJ program that can identify possible memory-safety violations at each point in the execution. The whole process has been automated with tool support and compared against other techniques. A worst-case analysis is performed that can give false negatives.

Contents

List of Figures

Acknowledgements

I would like to express my sincere thanks to my supervisor, Professor Ana Cavalcanti, for her continued support and guidance throughout my research. She has always been available to give help and advice when needed, for which I am extremely grateful.

I am grateful to Professor Andy Wellings, my internal examiner, for his feedback at each milestone of my PhD. I am also much obliged to Dr. Phil Brooke for taking the time to examine this thesis.

Thanks to the EPSRC for providing the necessary funding that allowed me to undertake this research. Thanks also to the Department of Computer Science, which has been a huge part of my life for the last 8 years throughout my PhD and undergraduate masters degree. I would also like to thank my colleagues in the department that have supported me. In particular, Dr Frank Zeyda, who has repeatedly offered his guidance and knowledge in formal methods and Safety-Critical Java.

On a personal level, I would like to thank my wife, Nicci, for her continued patience, understanding, and support. Thanks also to my friends, who have made this chapter of my life so enjoyable.

Finally, I would like to dedicate this work to my parents, for always being there for me. My father has always been my inspiration for completing this work, and without him I would not be where I am today. None of this, however, would have been possible if it was not for my mother, to whom I am eternally grateful.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2010 - 2014. Except where stated, all of the work contained within this thesis represents the original contribution of the author.

Some parts of this thesis have been published in conference proceedings; where items were published jointly with collaborators, the author of this thesis is responsible for the material presented here. For each published item the primary author is the first listed author.

- Chris Marriott, Ana Cavalcanti. SCJ: Memory-safety checking without annotations. In *Proceedings of the 19th International Conference on Formal Methods*, pages 465-480, May 2014. [28]
  The development of Safety-Critical Java (SCJ) has introduced a novel programming paradigm designed specifically to make Java applicable to safety-critical systems. Unlike in a Java program, memory management is an important concern under the control of the programmer in SCJ. It is, therefore, not possible to apply tools and techniques for Java programs to SCJ. We describe a new technique that uses an abstract language and inference rules to guarantee memory safety. Our approach does not require user-added annotations and automatically checks programs at the source-code level, although it can give false negatives.

Chapter 1

Introduction

This chapter introduces the work and gives the necessary background and motivation in Section 1.1. Section 1.2 describes the objectives of the work and includes the thesis hypothesis. Section 1.3 summarises the contributions made before Section 1.5 describes the outline of the thesis.

1.1 Background and motivation

Safety-critical systems are used, and relied upon, by everyone in today's society. The expectation on systems to automatically keep us safe is forever growing; recent examples include the introduction of pedestrian detection and automatic braking systems in cars [48], and automatic lane-departure warning systems [47]. Systems such as these, whose failure may cause serious injury or even death, are subject to extensive verification and certification processes, especially in the automotive and avionics industries, to try and ensure failure is not an option.

The Java programming language is undoubtedly one of the most common and popular programming languages for program developers today. Java is an object-oriented language, and object orientation is considered to be the dominant programming paradigm currently [9]. The language provides both compile-time and run-time checking; virtual machines make the language architecture independent, whilst features such as Just-In-Time (JIT) compilation gives better performance for specific environments. Java's ability to express concurrent implementations with threads also gives it appeal over languages such as C.

In the Java memory model, all objects are placed on the heap; local variables are stored on a method's stack. When all references to a particular object are removed from the run-time environment, an automatic garbage collector removes the object from the heap on its next execution. The Java memory model is very much behind-the-scenes to the developer; it is not necessary to think about where and when memory is allocated or deallocated. The deallocation of memory by the garbage collector happens automatically at potentially random points. This lack of control over the garbage collector provides a good abstraction for programmers, but presents a potential problem when running time-critical applications. Consider, for example, the automatic braking system previously mentioned; it is not acceptable for the automatic brakes to wait whilst the garbage collector operates.

Verification and certification are timely and costly procedures; methods to automate or facilitate these activities are an interesting topic of research for both academia and industry. One of the more recent attempts to aid the design, verification, and certification of safety-critical systems is the introduction of Safety-Critical Java (SCJ) [46]. An international effort, by various collaborators, has produced a specification for Java that is suitable for safety-critical and real-time Java programs. It is no surprise that Java is being adapted for use in safety-critical systems as it is already a widely used and popular object-orientated language.

The Real-Time Specification for Java (RTSJ) [6] makes Java more suitable for real-time systems, and provides both timing and behaviour predictability. RTSJ has been successfully applied in several real-world examples including a controlled UAV from Boeing and Purdue, and a battleship computing environment from IBM and Raytheon [33]. The guarantees of reliability needed for safety-critical systems, however, were hard to achieve without a further restricted language. SCJ strikes a balance between popular languages (such as Java and C), and languages already considered suitable for high-integrity systems (such as Ada).

This work is aimed specifically at the SCJ language as it is a new and upcoming language that has already received interest from both industry and academia. Its constrained memory and programming models make it a potentially tractable language in terms of static verification techniques.

The work is focused on memory safety of SCJ programs: the SCJ memory model is one of the distinguishing features that sets the language apart from the RTSJ and standard Java languages. The RTSJ introduces the notion of scoped memory areas that are not subject to garbage collection, however, the heap remains available for the programmer to create and reference objects with no additional concerns. The SCJ memory model goes one step further by completely removing access to the heap and limiting the entire program to scoped memory areas and immortal memory. In addition, it restricts the way in which the scoped memory areas are constructed and used.

The strict memory model of SCJ introduces the possibility of scoped memory violations that must be checked. It is not enough, like in standard Java, to suggest that the lack of null-pointers and array-out-of-bounds exceptions give a memory safe program. The definition of memory safety in the context of SCJ must be enriched to include the scope rules defined in the language specification. Briefly, the scoped memory areas in SCJ form a hierarchy, and it is not valid to reference an object that is stored in a child memory area as the object may be cleared out of memory before the reference variable.

SCJ programs are classified at a specific level, which corresponds to the level of complexity of the program. Level 0 programs are the most simple and are cyclic executive programs. Level 1 programs introduce concurrency and handle asynchronous events. Finally, Level 2 programs are the most complex and contain an even greater degree of concurrency as will be explained later. Level 0 programs will generally be reserved for the strictest of programs with tight deadlines and a high level of assurance. Level 1 programs are not as restricted, but will provide some of the more interesting examples used in safety-critical applications; therefore, Level 1 programs are the focus of this work.

As the SCJ language is relatively new, verification tools and techniques are currently fairly sparse, however, the technique in [45] verifies that a given SCJ program is valid according to the rules imposed by user annotations using a static checking tool. These annotations are used to define level, behavioural, and memory properties for particular classes and methods.

Another technique presented in [14] uses a bytecode checking technique to build a points-to model of a program that enables memory-safety analysis to be performed without user-added annotations. Bytecode analysis suffers from issues such as traceability and unnecessary false negatives being raised through the simplifications made at compilation.

The basic memory model of SCJ has been captured formally in the UTP in [11]. The memory model provides a basis to develop a formal representation of the necessary SCJ components required to verify that a given program is memory safe. A formalisation of the full language is not necessarily required in order to verify memory safety; abstractions can be made as will be discussed later in Chapter 3.

Work is also ongoing into the expression of a new variation of the process algebra *Circus*; it is being designed specifically to capture the SCJ programming paradigm independently of the code [12]. Using this language, the plan is for the development of a refinement strategy from abstract models, which do not consider the programming paradigm, to a more concrete representation that facilitates the automatic generation of SCJ programs. This work is complementary to that outlined here.

1.2 Objectives and hypothesis

In order to check memory properties of SCJ programs, it is important to understand what memory safety is and what restrictions are imposed by the SCJ specification. This work investigates the different memory models of Java, RTSJ, and the SCJ programming languages, and what it means for these to be memory safe. The advantages and disadvantages of existing memory checking methods need to be identified and addressed in the technique presented here.

Having identified existing techniques, and the restrictions imposed by each, the main objective of this work is to identify a new technique that does not suffer from the same restrictions whilst providing a method that is both useful and realistic for real-world applications. The aim is to develop a static checking technique that can identify memory-safety violations to prevent run-time exceptions.

Checking SCJ programs is not trivial, especially when all factors such as memory and timing are considered. It is not crucial, however, to analyse all aspects of a program when checking a specific property, and as such, abstractions can be made to make the checking process easier. Part of this work aims to identify the abstractions that can be made to simplify the analysis technique whilst maintaining all of the necessary information to reason about memory safety.

Another objective of the work is to identify memory safety rules that describe what it means for a program to be memory safe. If these rules can be applied to a program successfully, then no memory-safety violations exist.

With all of these objectives in mind, the following hypothesis is defined to summarise the aims of the work and outline what is believed to be possible.

It is possible to produce a sound, automatic static checking technique for valid Level 1 Safety-Critical Java programs to identify possible unsafe uses of mem-

ory at the source-code level, without the need for additional user-added annotations.

1.3 Contribution

The main contribution of this thesis is a new static-checking technique that guarantees to find possible memory-safety violations for Level 1 SCJ programs without the need for user-added annotations. In addition to the definition of the overall technique, an underlying formalisation that provides the basis to prove the soundness of the approach has also been developed.

Another contribution is found in the way methods are handled. In this technique, method properties, which are essentially method assertions or postconditions, are used to define the behaviour of the method independently of the calling context. The contribution is, therefore, an approach to define and analyse the behaviour of methods in the context of memory safety that supports modular reasoning.

The technique defined here is based on the analysis of an intermediate language, which is an abstraction of the original SCJ program. As the intermediate representation is an abstraction, and not a transformation of the original program, it is easier to maintain traceability in comparison to other existing techniques. The ability to map potential errors back to a specific statement in the original SCJ program is, therefore, another contribution of this work.

The intermediate representation has simple commands and expressions, and does not contain any nested or complex statements. This allows for the definition of simple memory-safety rules on the intermediate representation, which are easy to understand and follow. The fact that the rules are simple and easy to understand is a contribution in its own right. We view SCJ not only as a profile for Java, but a paradigm for programming of safety-critical systems that can be adopted in the context of other programming languages or even used to design new languages. Any such work can benefit from the rules we have presented, given the clear and abstract nature of the language for which they are described.

The static checking technique described here has been implemented in a tool and applied to several specific examples that generate memory-safety violations and also a number of case studies from the literature. The implementation of the tool, and the successful identification of possible memory-safety violations in examples, proves that the overall approach is capable of fulfilling the hypothesis. The tool is, therefore, a contribution as it allows others to verify that their SCJ programs are free of memory-safety violations.

Finally, a smaller yet valid contribution is the use of Z to formalise the entire approach, which provides the community with another case-study in the language.

The next section summarises the overall approach at a high level.

1.4 Overview

The technique has three main steps, as shown in Figure 1.1. The first step takes a valid SCJ program that is type correct and well formed according to the SCJ specification, and automatically translates it into the new language called *SCJ-mSafe*, which is designed to ease verification. This uses a translation strategy that has been formalised in Z. No information relevant to memory safety is lost, but all irrelevant information is discarded. Each SCJ program is described in the same style when translated to *SCJ-mSafe*; this makes programs easier to read and facilitates the analysis. A uniform structure also eases formalisation of *SCJ-mSafe* and of the checking technique, which is crucial in proving soundness.

In the second step, an analysis strategy, which has also been formalised in Z, is used to automatically generate method properties for each method in the *SCJ-mSafe* program. These method properties are calculated independently of the execution of the program, and give a summary of the method's behaviour. These properties are used during the analysis of the overall program at each method call; the allocation context at the calling point of the method gives meaning to the method properties and memory safety can be checked.

Finally, in conjunction with the method properties, inference rules are applied to the *SCJ-mSafe* program using an environment that is automatically constructed to capture memory properties of expressions required to determine memory safety. Each component of an *SCJ-mSafe* program has an associated rule that defines in its hypothesis the conditions that must be true for it to preserve memory safety. If all hypotheses of all rules applied to a program are true, then the program is memory safe. If any of the hypotheses are false, there is a possibility of a memory-safety violation.

Given an SCJ program, the technique consists of automatically translating it into *SCJ-mSafe* and applying the memory-safety rules. In this way, safety can be verified without additional user-based input such as annotations, for example.

In general, the memory configuration at particular points of a program cannot be

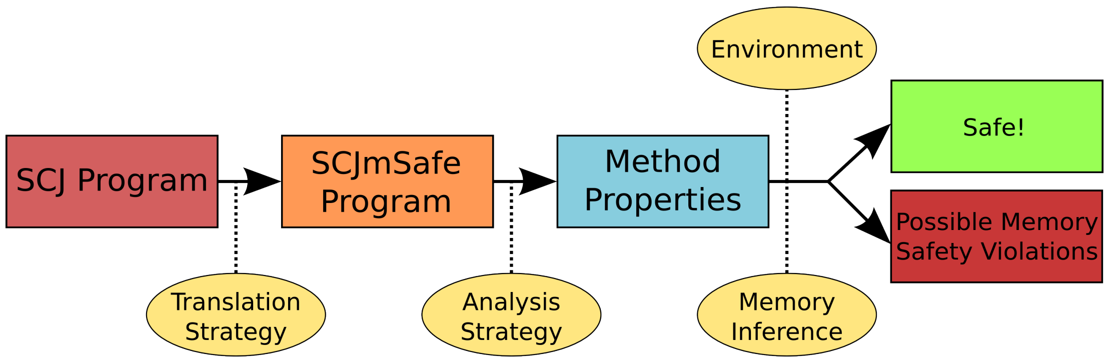


Figure 1.1: Memory-safety checking technique

uniquely determined statically. It may depend, for example, from the values of inputs to the program. Since the aim is to perform a static analysis, the worst-case scenario is always assumed when checking memory safety.

The analysis is flow sensitive, path insensitive, context sensitive, and field sensitive. The flow of the program is considered by checking each command individually as opposed to summarising behaviour.

Precise knowledge of the control path is not necessary. For example, which branch of a conditional statement is executed cannot be determined statically; both branches are considered. Although the behaviour may be different in each branch, the effect on memory may be the same; if not, the effects of both branches are considered. Analysis of loops and recursion is based on the calculation of a loop summary.

The analysis is context sensitive as methods are analysed based on their calling site, although each method is analysed once to establish a parametrised summary of behaviour. This summary is used at each calling point of the method.

Finally, the analysis is field-sensitive as it considers all fields of a referenced object when analysing assignments and new instantiations.

1.5 Thesis outline

Chapter 2 gives an introduction to Java, RTSJ, and SCJ, and describes the differences between the memory models of each. It justifies the need for real-time and safety-critical variations of Java, and how the different memory models are applicable. It also discusses the existing memory checking techniques for RTSJ and SCJ programs, and evaluates their effectiveness and limitations.

Chapter 3 describes the intermediary language called *SCJ-mSafe* language, which gives an abstract representation of SCJ programs. *SCJ-mSafe* is designed to ease verification as all programs are expressed in the same way; each program is a sequence of definitions of components of an SCJ program. It also describes the translation strategy from SCJ to *SCJ-mSafe*.

Chapter 4 describes the components required to perform analysis on *SCJ-mSafe* programs. An environment is defined that holds information about reference variables and their corresponding objects. The chapter also describes method properties, which capture the behaviour of methods independently of the calling context. The memory safety inference rules that define what it means for each *SCJ-mSafe* component to be memory safe are also presented. Finally, the chapter describes how the environment is checked every time it is updated in order to give a precise location in the program where a possible memory-safety violation may occur.

Chapter 5 introduces the tool support for the technique that is capable of automatically translating and checking a given SCJ program for possible memory-safety violations. It demonstrates how specific programming patterns that are known to introduce possible memory-safety violations are handled, and includes examples of each to illustrate the checking technique and the output of the tool. The chapter also includes details of several case studies that have been successfully checked with the automatic checking tool.

Finally, Chapter 6 draws some conclusions about the technique, and describes in more detail the possible future work to be completed. In particular, how the technique could be extended in order to work with Level 0 and Level 2 SCJ programs, and how the soundness of the technique could be proven.

Chapter 2

Memory safety of real-time and safety-critical Java programs

Memory safety refers to the property of a program whose execution is free from run-time errors from dangling references. This chapter introduces RTSJ and SCJ and their different memory models; it also describes what it means for a program written in each of these languages to be memory safe, and how existing techniques have been developed to verify this property.

Section 2.1 introduces the RTSJ and SCJ languages. Section 2.2 describes the memory models of these languages, as it is memory safety in particular that this thesis is concerned with. Section 2.3 discusses existing techniques for checking memory safety of RTSJ programs, whilst Section 2.4 explains techniques for SCJ. Finally, Section 2.5 summarises the existing techniques and gives motivation for the technique presented in the remaining chapters.

2.1 Real-Time and Safety-Critical Java

In order to make Java more applicable for use in real-time and safety-critical programs, programming styles and new language features have been specified in two different Java variants: RTSJ, and SCJ. These two languages are discussed in more detail below.

2.1.1 Real-Time Specification for Java

The RTSJ was designed to address the limitations found in standard Java when developing real-time programs. The idea was to create a language that imposes as few limitations on the developer as possible, whilst also giving them the functionality required to express real-time properties. The main additions to standard Java, found in the RTSJ, are discussed individually below [49].

Time The standard concept of calendar time provided by Java is not enough for time-critical systems. RTSJ introduces high-resolution time, which has granularity of a nanosecond. This concept of time is then extended into two main categories: relative, and absolute. Relative time is a simple duration from one point in time to another. Absolute time defines an exact fixed point in time.

Scheduling In standard Java, the user has no guarantees about scheduling in the JVM; this is not acceptable for systems with priority-based schedulables. The RTSJ supports pre-emptive priority based scheduling. All schedulable objects in RTSJ have three parameters: a release requirement, a memory requirement, and a scheduling requirement. The release requirement defines when the schedulable object is ready to run. The memory requirement defines the rate at which the object allocates memory. Finally, the scheduling requirement defines the priority of the object.

Schedulable objects can be periodic, aperiodic, or sporadic. Periodic event handlers have a fixed arrival frequency whereas aperiodic and sporadic event handlers do not; they are often triggered by external inputs to the system. The difference between aperiodic and sporadic is the minimum inter-arrival time found in sporadic events, which specifies the minimum time that must pass before the event can occur again.

Threads Java contains threads; the RTSJ introduces real-time threads, which inherit the same requirements of a schedulable object. Real-time no-heap threads are also real-time threads, but they guarantee not to reference or allocate any objects on the heap; this makes them independent of the garbage collector.

Asynchronous events Threads are often used to perform tasks that are not associated with some specific event; for this we use asynchronous event handlers.

RTSJ has been successfully applied in several real-world examples including a controlled UAV from Boeing and Purdue, and a battleship computing environment from IBM and Raytheon [33]. However it is often the case that restrictions have been applied to facilitate analysis of RTSJ programs; the development of Safety-Critical Java, which is

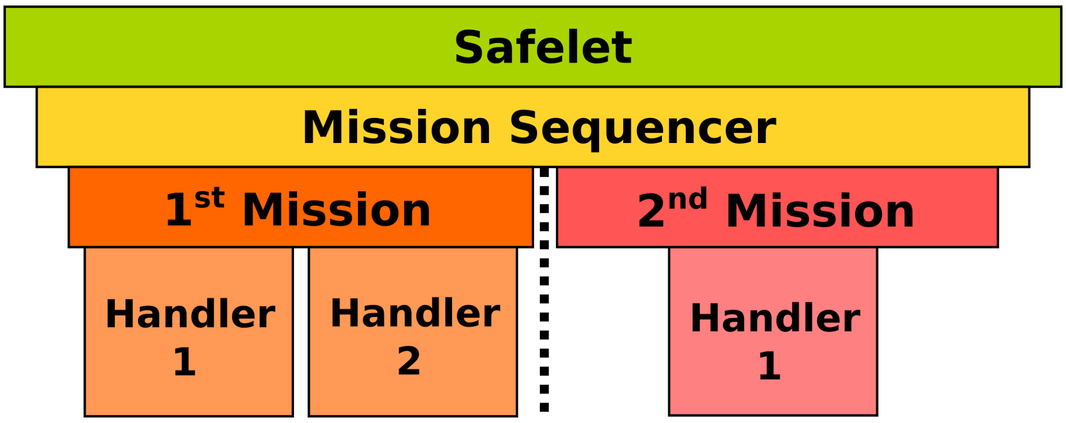


Figure 2.1: SCJ programming paradigm.

presented in the next section, was undertaken to try and make Java more suitable for safety-critical applications.

2.1.2 Safety-Critical Java

The SCJ specification is an official Java Specification Request (JSR-302) currently under development by The Open Group. The latest draft version of the specification (v0.97) is publically available, and was released in June 2014. The work presented here is based on v0.94 from June 2013, and is referred to as 'the specification' from this point onwards [46].

The specification is based on Java and the RTSJ; it is designed to be more suited to safety-critical systems, and in particular their certifiability. Restrictions are often imposed to ensure that programs are suitable for certification; an example is garbage collection, which is not considered suitable for use in real-time and safety-critical systems. The memory model of SCJ, which does not use garbage collection, is one of its most distinguishing features, and is discussed in more detail below.

SCJ programs that conform to the specification, and use safety-critical libraries, may be certifiable to Level A of the DO-178B [37] avionics standard. Level A systems are defined as those whose failure could lead to catastrophic consequences and subsequently prevent an aircraft from continuing safely. SCJ programs written to the specification are not automatically certifiable; the specification is designed to make certification easier by providing a programming paradigm.

SCJ Paradigm The SCJ programming paradigm is focused around the concept of missions, where each mission has a number of event handlers. Figure 2.1 gives a graphical representation of the paradigm, and shows the four fundamental components: the safelet,

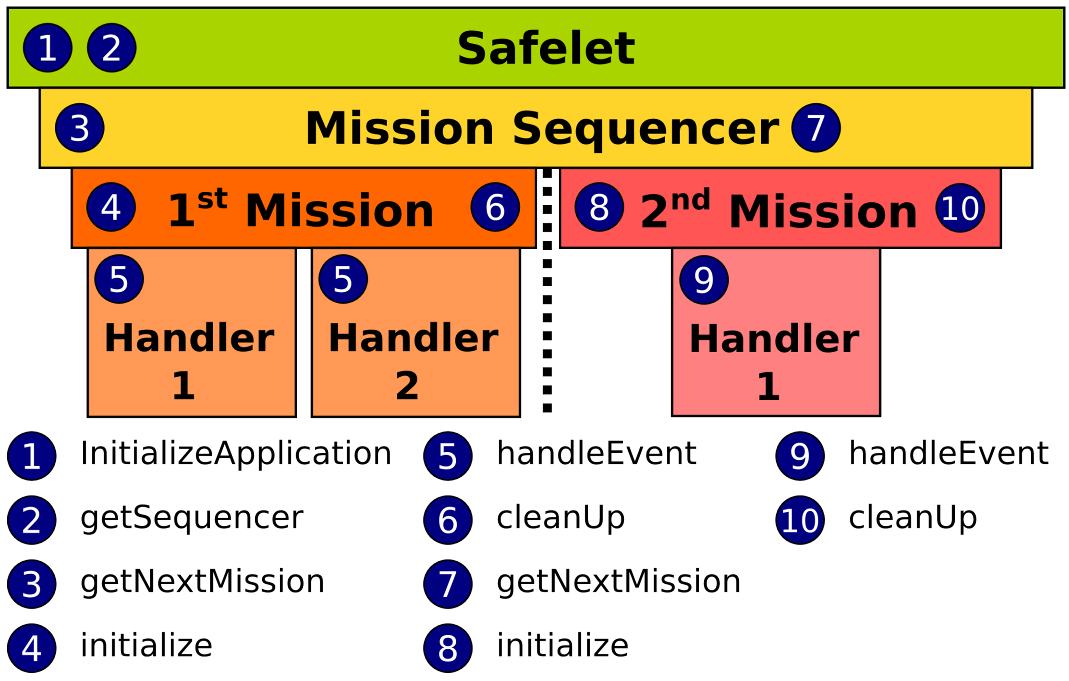


Figure 2.2: SCJ programming paradigm with execution flow.

mission sequencer, missions, and handlers. The dotted line in between the two missions represents a sequence, as only one mission can execute at a time per mission sequencer.

Figure 2.2 shows the execution flow of an SCJ program inside the paradigm. The entry-point for an SCJ program is the safelet, which performs the necessary setup procedures of the program (1) before creating a mission sequencer (2). The mission sequencer controls the sequence of missions to be executed (3). Missions have three phases of execution: `initialisation` (4), `execution` (5), and `cleanUp` (6). Periodic and aperiodic handlers are pre-allocated during the initialisation phase of a mission (4), before executing in the execution phase (5); a handler's execution is based on its `handleEvent` method. Finally, once the event handlers of a mission have finished executing, the cleanup phase (6) is entered to perform any final tasks before the mission finishes. The mission sequencer is then responsible for returning the next mission to execute (7), and the cycle continues.

Compliance levels Compliance levels are used to define the complexity of a program. For example, hard real-time applications are often likely to contain a single thread of execution with simple timing properties to ensure deadlines are not missed. More complex programs may be highly concurrent with multiple threads executing at the same time. Accordingly, SCJ has three compliance-levels: 0, 1 and 2. Level 0 programs refer to the most simple programs described above, whilst Level 2 programs have increased complexity.

- **Level 0** programs are cyclic executive programs. Missions contain only periodic event handlers, which have fixed periods, priorities, and release times (in relation to a cycle). There is no concurrency at this level. Only sequential missions are allowed under a single mission sequencer.

- **Level 1** programs introduce aperiodic and one-shot event handlers. These, along with periodic event handlers, are executed concurrently in each mission. Schedulable objects are controlled by a fixed-priority pre-emptive scheduler. Missions remain sequential at this level under a single mission sequencer.

- **Level 2** programs are the most complex and introduce real-time no-heap threads. They also allow concurrent missions and nesting of missions; this is achieved through multiple mission sequencers executing in parallel underneath the top-level mission sequencer. Methods that may cause blocking, such as `Object.wait` and `Object.notify` are also allowed at this level.

Level 2 programs may be hard to analyse because of the possibility of multiple separate missions executing concurrently. As such, existing research into SCJ has predominantly focused on Level 1 applications, which is considered similar in complexity to Ravenscar Ada [8, 11]. The work presented here is based on Level 1; the motivation for Level 2 SCJ programs, and details of the added complexity, is discussed further in [50].

Annotations The SCJ specification includes specific annotations to express constraints on programs. These annotations facilitate static analysis to be performed to ensure implementations conform to the specification rules. They are also maintained in compiled bytecode to allow checks at class load-time. The annotations are split into two categories: compliance-level and behavioural.

Compliance-level annotations are used to ensure classes and methods are only used at the correct level. For example, a Level 1 implementation could not use a method defined as Level 2 compliant because the behaviour may break the restrictions of Level 1 programs; however, a Level 2 implementation could use a Level 1 method. Behavioural annotations are used to restrict properties such as blocking and allocation.

The annotations mentioned above are discussed in more detail in Section 2.4, which also includes a set of proposed memory-safety annotations for SCJ programs.

```
public interface Safelet
    public void initializeApplication()
    public MissionSeqencer getSequencer()

public abstract class MissionSequencer
    protected abstract Mission getNextMission()

public abstract class Mission
    protected abstract void initialize()
    public MissionSequencer getSequencer()
    protected void cleanUp()
    public static Mission getMission()

public abstract class PeriodicEventHandler
    public void register()

public abstract class AperiodicEventHandler
    public void register()
    public final void release()
```

Figure 2.3: Concise SCJ API

SCJ API A concise version of the SCJ API that illustrates the programming paradigm and the SCJ-specific methods in each component is shown in Figure 2.3. The execution stages described in Figure 2.2 correspond to the methods of the API.

In order to demonstrate the SCJ language, and illustrate how programs are developed with the new programming paradigm, the next section describes a cruise control system that has been implemented in SCJ. The next chapter will demonstrate how this example is expressed in the intermediate representation used for the static checking technique.

2.1.3 A Cruise Controller System

Figure 2.4 (taken from [54]) shows the class diagram for an Automotive Cruise Controller System (ACCS), which automatically monitors and maintains the speed of a vehicle. The diagram shows how the program is structured according to the SCJ paradigm; it also shows the SCJ infrastructure classes that are extended by each component in the implementation.

The example was originally described in [49], and an implementation as a Level 1 SCJ program has been described in [53]. The implementation has a single mission, and is made up from seven handlers that monitor the vehicle's gears, engine, brakes, throttle, levers, wheel shaft, and speed. When an event occurs, such as pressing the brake pedal, the associated handler communicates with the overall controller class in order to maintain

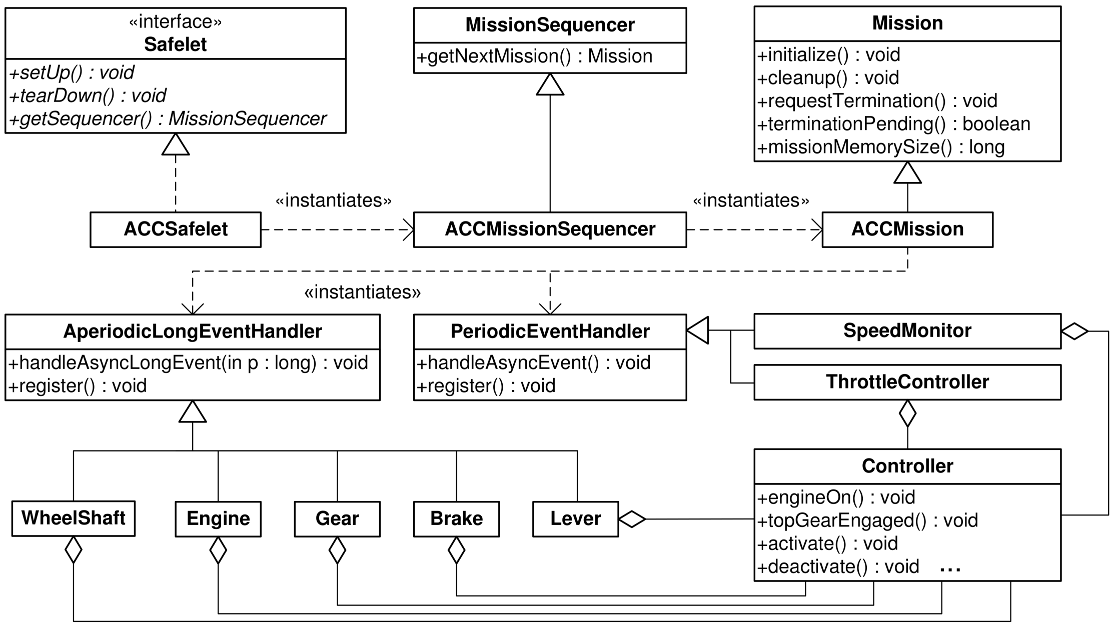


Figure 2.4: ACCS class diagram

an accurate representation of the vehicle's state. In this situation, for example, the cruise control system is automatically disabled because the driver of the vehicle has pressed the brake pedal. A more detailed explanation of the system can be found in [53]; in what follows, parts of the code are explained as needed.

Part of the ACCS mission class in SCJ is shown in Figure 2.5. It includes the `initialize` method on line 5, which instantiates and registers all of the handlers in the system. The `createEvents` method call on line 6 creates the events that correspond to interactions between the vehicle and the cruise controller; these are real-world events, such as the triggering of a sensor, for example. The `createISRs` method call on line 7 creates the corresponding interrupt service routines (ISR) for each event; these are responsible for firing the aperiodic handlers when the interrupt occurs. These ISRs are then registered through the `registerISRs` method call on line 8.

The `WheelShaft` handler created on line 11 monitors the rotation of the wheel, and increments a local counter when fired. A reference to the `WheelShaft` handler is passed to the `SpeedMonitor` handler created on line 12, which periodically monitors the number of rotations recorded by the `WheelShaft` handler, and calculates the current speed. The `SpeedMonitor` handler is passed to the `ThrottleController` handler created on line 13, which is responsible for maintaining the speed of the vehicle, since controlling the speed requires knowing how fast the vehicle is currently going.

```
1  class ACCMission extends Mission {
2
3    ...
4
5    public void initialize() {
6      createEvents();
7      createISRs();
8      registerISRs();
9
10     /* Create handlers and controller. */
11     WheelShaft shaft = new WheelShaft(shaft_event);
12     SpeedMonitor speedo = new SpeedMonitor(shaft, 500);
13     ThrottleController throttle = new
           ThrottleController(speedo);
14     Controller cruise = new Controller(throttle, speedo);
15     Engine engine = new Engine(cruise, engine_event);
16     Brake brake = new Brake(cruise, brake_event);
17     Gear gear = new Gear(cruise, gear_event);
18     Lever lever = new Lever(cruise, lever_event);
19
20     /* Register event handlers with the mission. */
21     shaft.register();
22     engine.register();
23     brake.register();
24     gear.register();
25     lever.register();
26     speedo.register();
27     throttle.register();
28   }
29 }
```

Figure 2.5: ACCS Mission in SCJ

The `Controller` class created on line 14 is not a handler, it is responsible for maintaining the state of the system. It is passed as a reference to the subsequent `Engine`, `Brake`, `Gear`, and `Lever` handlers created on lines 15-18 as they can all invoke methods in the `Controller` class that change the state of the system. Finally, all of the defined handlers are registered on lines 21-27.

Figure 2.6 shows the `SpeedMonitor` handler, which has several fields to record the necessary information to successfully calculate the current speed of the vehicle. The constructor of the handler includes a call to the parent constructor via the `super` method on line 12; this records the necessary information for scheduling and memory usage. The first parameter to the `super` call is the priority of the scheduler, which in the case of this handler, is the highest possible priority in the system; only the throttle controller shares the same highest priority. The second parameter determines the frequency at which the periodic handler should be fired; this is passed as a parameter from the instantiation in the

```
1  public class SpeedMonitor extends PeriodicEventHandler {
2    public final int calibration; /* cm per rotation */
3    public final int iterationsInOneHour;
4    public final int cmInKilometer = 100000;
5
6    private WheelShaft wheel_shaft;
7    private long numberRotations = 0;
8    private long lastNumberRotations;
9    private int currentSpeed = 0; /* kilometers per hour */
10
11   public SpeedMonitor(WheelShaft shaft, long period) {
12     super(
13       new PriorityParameters(
14             PriorityScheduler.instance().getMaxPriority()),
15       new PeriodicParameters(null, new RelativeTime(period,
            0)),
16       new StorageParameters (32768, 4096, 4096),
            "SpeedMonitor");
17
18     calibration = wheel_shaft.getCallibration();
19     iterationsInOneHour = (int) ((3600*1000) / period);
20     wheel_shaft = shaft;
21     lastNumberRotations = wheel_shaft.getCount();
22   }
23
24   public synchronized int getCurrentSpeed() {
25     return currentSpeed;
26   }
27
28   public void handleAsyncEvent() {
29     numberRotations = wheel_shaft.getCount();
30     long difference = numberRotations - lastNumberRotations;
31     currentSpeed = (int) ((difference * calibration *
          iterationsInOneHour) / cmInKilometer);
32     lastNumberRotations = numberRotations;
33   }
34 }
```

Figure 2.6: SpeedMonitor Handler in SCJ

mission class, and is 500ms. Finally, the third parameter describes the amount of memory that is required for the handler to execute. The remaining commands in the constructor set up the necessary calibration information to calculate the speed; this is based on the wheel shaft information and the frequency of the calculation.

When the handler is fired, the `handleAsyncEvent` method defined on line 28 is called; this gets the current number of wheel rotations from the `WheelShaft` handler, and calculates the difference based on the previous execution. The current speed is then calculated, and the number of rotations updated.

This example demonstrates the new programming paradigm introduced by SCJ, and in particular illustrates the roles of periodic and aperiodic event handlers within a system. The controller class, which is instantiated inside the mission memory area is passed as a reference to multiple event handlers, so that information can be transferred between the mission and handlers. Passing references to objects between different memory areas could potentially lead to the introduction of memory-safety violations, however, there are non in this example. The different memory models of SCJ and the RTSJ are presented in more detail next.

2.2 Memory models and memory safety

This section describes the memory models of the Java, RTSJ, and SCJ programming languages. It also defines what it means for an SCJ program to be memory safe.

2.2.1 Java memory model

In the Java memory model, all objects are placed on the heap. Local variables are stored on the stack; these variables may be primitive values, or reference values to objects. When a scope finishes, the respective fragment of the stack is removed, and all local variables stored are no longer accessible; this does not automatically remove objects in the heap referenced from the stack. Once an object becomes unreachable in the program, it is removed from the run-time environment, Java's automatic garbage collector removes the object from the heap and frees up the memory on its next execution.

The memory model is behind-the-scenes from the point of view of the developer; it is not necessary to think about where and when memory is allocated or deallocated. In fact, the deallocation of memory by the garbage collector happens automatically, and occurs at potentially random points. There is no explicit allocation and deallocation of memory as found in languages such as C, for example.

The lack of predictability from the garbage collector presents a potential problem when running time-critical applications. To address this problem, and make Java more suitable for use in real-time applications, the RTSJ language adds a more suitable memory model.

2.2.2 RTSJ memory model

As mentioned previously, it is not a desirable property for real-time applications to be interrupted by the garbage collector, particularly if the point of interruption cannot be predicted. To overcome this problem, the RTSJ language introduces a region-based memory model. These regions store dynamically created objects, and are not subject to garbage collection. The use of regions gives the developer control of the memory structure knowing that objects in regions are reclaimed together.

The two types of memory regions used in RTSJ are immortal and scoped areas, in addition to the heap. The immortal memory region exists for the entire length of the program; objects stored there cannot be removed, and remain in the memory until the program terminates. In contrast, scoped memory areas have a shorter lifetime; all objects stored in a scoped region are reclaimed when the region's lifetime is finished. This is determined by the number of schedulable objects executing in the region; a schedulable object executes inside a region when the object is active and its allocation context is the region in question. Once there are no schedulable objects executing in the region, the objects are reclaimed. The memory region, which is an object itself, is reclaimed once its containing region, or parent region, has no schedulable objects executing and is cleared out.

Scoped regions can be defined and entered explicitly, or assigned to schedulable objects; this provides a technique for assigning individual memory areas that are not at risk of interruption from the garbage collector to runnable objects. Scoped memory areas can be nested, creating a hierarchical cactus-like structure of memory regions. This can potentially cause errors with references to objects that resides further down the memory structure than the reference variable. To ensure this problem does not arise, the following rules exist for the RTSJ memory model:

- Objects in the heap cannot reference objects stored inside scoped memory areas.
- Objects in immortal memory cannot reference objects stored inside scoped memory areas.
- Objects in scoped memory areas can only reference other scoped memory areas if the target area is down the scope cactus, that is, towards the root.
- Scoped memory areas must have only one parent, that is, the outer memory area in which the new memory area was entered.

RTSJ also includes a heap memory region, which behaves like the heap in standard Java; the heap is subject to garbage collection, unlike the scoped memory regions. An example

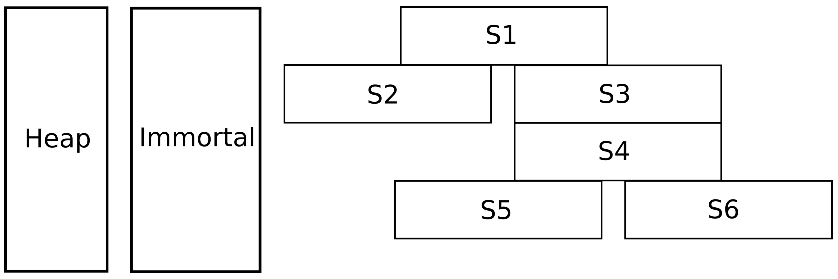


Figure 2.7: RTSJ memory structure

of a memory configuration of an RTSJ program is shown in Figure 2.7. It shows the heap and immortal memory in conjunction with a cactus-structure of scoped memory areas. The immortal memory area is separate as it lasts for the entire length of the program; scoped memory areas S1 to S6 are created throughout the program execution. It is possible to split the cactus such that memory areas have different paths to the root (like that of S1, S2, and S3 in the diagram); this is because it is possible to execute code in existing memory areas. It is not possible, however, to create cycles in the cactus, as this invalidates the single-parent rule associated with memory areas.

2.2.3 SCJ memory model

The SCJ specification takes the memory restrictions one step further than RTSJ by completely removing the heap and garbage collector. Similarly, two types of memory area are defined: immortal, and scoped memory areas. Immortal memory is the same as that in RTSJ. Scoped memory areas are more restricted than those in RTSJ, and are used for individual aspects of the SCJ programming model.

The immortal memory area is the top-level memory area, and lasts for the duration of the program's execution. Missions have their own scoped memory, called mission memory, which lasts for the duration of a single mission. Handlers also have their own scoped memory area for the execution of their `handleAsyncEvent` method; these are called per-release memory areas, and last for the duration of the `handleAsyncEvent` method. The final scoped memory area is the temporary private memory area, which is created and entered by specific SCJ commands. Temporary private memory areas are used to execute runnable objects, and only last as long as the runnable object's `run` method. The SCJ memory structure contains individual thread stacks for the program, mission sequencer, and each event handler.

The hierarchy shown in Figure 2.8 demonstrates how temporary private memory areas

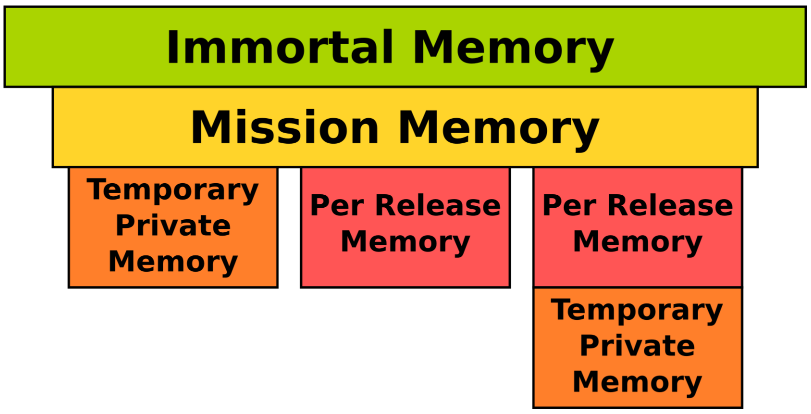


Figure 2.8: SCJ memory model.

can be entered from both the mission memory area (when in the initialisation phase of the mission) and the per-release memory area of a handler (during the execution phase of the mission). The hierarchy shown cannot be created directly at a specific point of execution in a program because of the different stages of execution in each mission, but it is designed to illustrate the relationship between scopes throughout all stages of execution in a program. The two per-release memory areas correspond to two handlers executing in the mission.

Each component of the SCJ paradigm is created in a specific memory area, and has a default allocation context (memory area); new objects created during execution are automatically created in these associated areas unless specified otherwise. Figure 2.9 shows the location of paradigm components and the default allocation contexts. The safelet and mission sequencer are created in immortal memory, and allocate new objects in immortal memory.

Missions, which are created as objects, reside in the mission memory area; the SCJ infrastructure creates and enters the mission memory area before the `getNextMission` method of the mission sequencer is called. New objects created by the mission are created in the mission memory area.

Event handler objects are also created in the mission memory area, during the initialisation phase of the mission. Once the execution phase is entered and a handler is fired, a per-release memory area associated with the handler is entered; new objects created during the `handleAsyncEvent` method are created inside the per-release memory area associated with the handler.

Temporary private scoped memory areas can be created and used during the initialisation phase of a mission, and by individual handlers; these are organised in a stack-based

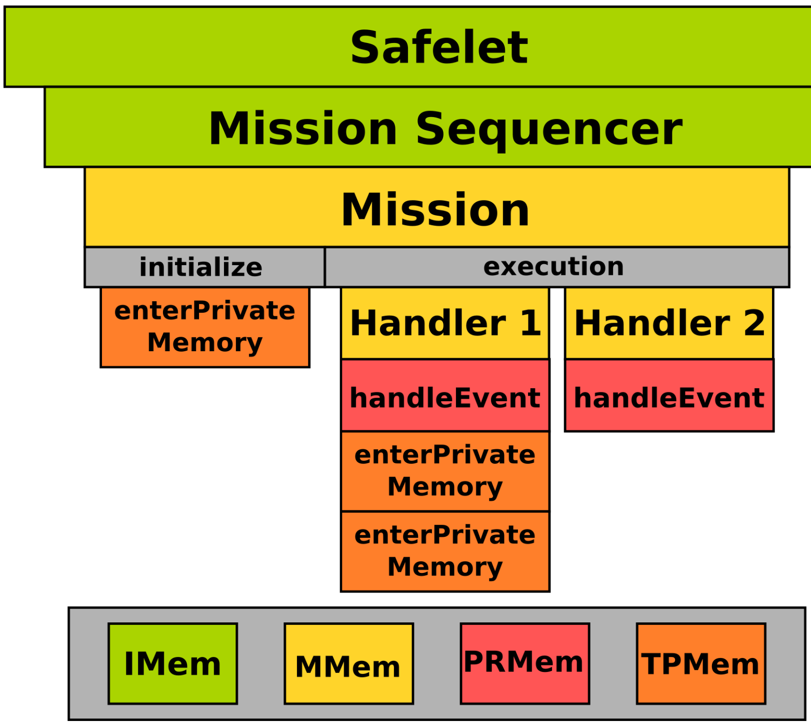


Figure 2.9: Default memory areas for SCJ paradigm components, and default allocation contexts.

structure. The default allocation context whilst the runnable object is executing is the current temporary private memory area.

The following SCJ-specific methods are capable of changing the default behaviour of the SCJ memory model, and introduce scope for memory errors.

- `newInstance` - This method can be called on a reference to a particular memory area, and creates a new object of the given type in the specific memory area. A reference to the new object is returned, which can be assigned to some expression, creating a reference from one memory area to another.

- `newArray` - Like the `newInstance` method, this method creates a new array object of a particular type in another memory area.

- `executeInAreaOf` - This method takes a runnable object and a reference variable, and executes the runnable object's `run` method in the memory area in which the reference variable resides.

- `executeInOuterArea` - Similarly to the `executeInAreaOf` method, this method takes a runnable object and executes its `run` method in the immediate outer memory area.

```
public abstract class MemoryArea
    public static MemoryArea getMemoryArea(Object object)
    public Object newArray(Class<> type, int number)
    public Object newInstance(Class<> type)

public final class ImmortalMemory extends MemoryArea

public final class ScopedMemory extends MemoryArea

public abstract class ManagedMemory extends ScopedMemory
    public static void enterPrivateMemory(long size, Runnable
        logic)
    public static void executeInAreaOf(Object obj, Runnable
        logic)
    public static void executeInOuterArea(Runnable logic)
```

Figure 2.10: Concise SCJ memory model API

- `enterPrivateMemory` - This method takes a runnable object and executes its `run` method in a new temporary private memory area.

The SCJ memory model API is shown in Figure 2.10. It illustrates how the memory areas of SCJ are defined by an abstract `MemoryArea` class, which encapsulates both the immortal and scoped memory areas. The SCJ-specific methods described above that have a direct impact on the possible memory safety of a program are also shown.

To avoid the possibility of dangling references, the SCJ memory model has strict rules about references within the hierarchical memory structure. References can only point to objects stored in the same memory area, or a memory area that is further up the hierarchy, that is, towards the immortal memory. For example, it is safe for a reference variable in the mission memory to point to an object in the immortal memory, because the immortal memory outlives the mission memory; therefore, the reference will never point to an object that does not exist. Alternatively, if a reference variable in the mission memory points to an object stored in the per release memory area of a handler, the reference may point to an object that has been reclaimed once the handler has finished executing; if referenced, this will produce a run-time exception.

2.2.4 Memory safety

The specific definition of memory safety for a particular language is defined based on the rules of its memory model. It is not unreasonable, however, to state that for any given programming language, a program is considered memory safe if there is no possible execution path through the program that could raise a memory-related run-time error.

Memory errors can be caused by several factors including buffer overflows, stack overflows, use of uninitialised variables, null pointers, or dangling references. The memory models of RTSJ and SCJ increase the possibility of dangling references through the use of scoped memory areas. Memory safety of references, therefore, corresponds to a program that has no possible execution path that violates the scoped memory laws of each language.

A program is typically defined as memory safe if there are no possible executions paths that may lead to an attempt to access an object via a dangling reference, that is, a reference to an object that has been previously deallocated.

In the context of RTSJ and SCJ, a more restrictive view is taken. Objects are not allowed to refer to objects in an memory area lower in the hierarchy. Such references may lead to dangling references and are forbidden altogether. Therefore, the following definition of memory safety is defined.

Memory Safety *A program is memory safe if there are no possible execution paths that may lead to an exception being raised by a reference variable attempting to point to an object in a memory area that is lower in the hierarchy, that is, further away from the immortal memory area.*

The focus of the work presented here is on memory safety; in particular, the definition given above. Other aspects of memory safety, such as checking for null pointers or array-out-of-bounds checking, for example, are out of the scope of this work. Whilst it may be possible to extend the approach to statically check for possible null pointers, these techniques have been well investigated in the literature and tools such as ESC/Java, for example, already exist to find possible errors in Java programs [16]. The next section describes existing techniques that have investigated memory safety of RTSJ programs.

2.3 Verifying memory safety in RTSJ

This section introduces existing techniques to check for possible memory-safety violations in RTSJ programs that have been described in the literature. Each section describes a different approach before some conclusions are drawn.

2.3.1 Memory management based on method invocation

The memory model of RTSJ introduces the possibility of programming errors and increases in execution time of an entire system. This is due to the additional control given to the programmer with respect to memory management, and the additional run-time checks required in RTSJ to ensure no dangling references occur in the memory structure. The guidelines outlined in [23] reduce these problems by removing explicit memory area control from the programmer, and by introducing a less complicated programming model. The authors explain that a well-defined programming paradigm eliminates the need for expensive run-time checks all together.

The programming language used in [23] is Ravenscar-Java [24], which is similar to Ravenscar-Ada [8]. It is a restricted language for use with single-processor systems and defines two execution phases: initialisation and mission phase. These phases, in conjunction with the memory model, produce a programming paradigm similar to that outlined in SCJ. The technique presented is defined as an optional extension to the Ravenscar-Java profile as it relaxes one of the original rules; that is, nested scoped memory areas are allowed, but not in the original profile.

The technique is based on the link between memory areas and methods; more specifically, each method in a program is assigned to an anonymous memory area. This removes the need for the programmer to understand the memory model, as the corresponding memory area is not known, and only decided at run-time. When a method is called, the associated memory area is entered; all objects created by the method are stored inside the memory area. Once the method has finished executing, the memory area is cleared out and reclaimed.

Threads do not have references to the memory areas in which they execute, and there can be at most one thread executing in a memory area at one time. This means that no thread can enter the same memory area twice; therefore, by using anonymous memory areas, the run-time checks that ensure the single-parent rule is enforced are no longer required.

References to objects created by a method in the same memory area are acceptable, as are references to objects created by methods in higher memory areas; that is, caller methods. References to objects that are created in a callee method are not allowed as they break the dangling reference rule. There are no references from scoped memory areas to the heap to prevent interaction with the garbage collector.

The technique improves performance of the run-time checks by adding additional information to the code that prevents unnecessary checks. For example, when a method returns a value that must be propagated up several nested method calls, a parameter that specifies where the return object is to be assigned is placed at the original caller method; exceptions are handled in the same way.

The additions to the code to aid run-time checks take the form of Java 1.5 annotations; their usage is described below.

- **ScopedMemoryMethod** - This annotation declares that a method is invoked with its own memory area. If this annotation is not used, methods use the memory area associated with the caller.
- **ReturnedObject** - This annotation declares the particular object that is returned to the caller.
- **PropagatedException** - This annotation is used to declare in which parent memory area exceptions are allocated.

The annotated Ravenscar-Java code can be used in one of two ways. It can be translated into RTSJ and used in conjunction with existing RTSJ virtual machines; or alternatively, a custom virtual machine that has knowledge of the annotations could be used. The authors favour the transformation technique to make use of the existing RTSJ tools as the semantics are the same.

The technique eliminates the need for the single-parent check, and improves the performance of other memory related run-time checks. Whilst this may prove useful for developers, as it simplifies the programming model of RTSJ, annotations are required to improve performance at run-time. A static checking technique that completely eliminates the need for memory related run-time checks would be even more efficient.

2.3.2 Type systems

Before the introduction of the SCJ specification, work was already being done to make RTSJ more suitable for use within safety-critical systems. Nilsen presents a type system to ensure scope safety in safety-critical Java modules [31], that is, reusable Java modules that are applicable for use within safety-critical applications and have been successfully certified. By using modules that are already certified, the need for whole-program analysis and certification is eliminated; this means that small changes to the developer code should not require re-certification or re-analysis.

Pre-certified safety-critical modules, designed as off-the-shelf additions to a development, must come with strict properties to ensure that varied use in different settings never raises errors. The technique in [31] describes a series of type attributes that apply to Java components through code annotations. These attributes are used to define memory properties that explicitly add information about the location of objects referenced by variables; these are described below.

- **Scoped** - Variables that may hold references to objects stored in a scoped memory area are given the scoped attribute. Any variable that does not have this attribute cannot reference objects in scoped memory.
- **Immortal** - Variables that hold references to immortal memory, or null, are given this attribute.
- **Array** - Variables that hold references to array objects stored in scoped memory areas have the array attribute; this is used in conjunction with the scoped attribute. Objects referenced by elements in the array are assumed to have the scoped attribute, unless specified otherwise.
- **Local** - Variables that hold references to objects stored in scoped memory areas have the local attribute if the variable is not assigned to another variable that may live longer than the specified variable. This is used in conjunction with the scoped attribute.
- **Result** - Variables whose value may be returned from a method are assigned the result attribute.

These attributes define a technique to determine what type of memory is used for variables; for example, reference variables with no attributes point to objects stored on the heap. By using this system, the technique does not restrict references to explicit memory areas, only the type of memory area used. For every assignment to a scoped variable, data-flow analysis is used to identify specific conditions that, if true, remove the requirement for a run-time check.

Programs are checked using Java bytecode verification to ensure the scope-safe type system is maintained. As mentioned in the attributes above, scoped variables contain references to objects in scoped memory areas, whilst unannotated variables do not. The verifier checks to make sure that objects referenced by scoped variables are never assigned to non-scoped variables. Similarly, the verifier ensures that no local scoped variable is assigned to another variable that does not have the local scoped attribute. The verifier maintains consistency of attributes throughout inheritance and overriding; more specifically, attributes declared in the superclass cannot be removed in subclasses.

Whilst this technique ensures that no objects are referenced by variables that might outlive the object, the number of annotations required is not insignificant. The method of explicitly defining whether an object resides in scoped memory or not does not impose a great restriction.

2.3.3 Ownership types

An alternative type system to provide a static checking technique for real-time Java combines the use of ownership and region types in [7]. The type system guarantees that well-typed RTSJ programs never fail at run-time through a memory-related software error, therefore eliminating the need to incorporate the additional run-time checks during execution, which in turn reduces the amount of overhead required.

Ownership types are used to enforce object encapsulation and allow modular reasoning; region types ensure that no dangling references are ever followed. For multi-threaded programs, the notion of subregions is introduced; subregions allow long-lived threads to share objects without utilising the heap or risking memory leakage. The use of subregions allows threads to create and reclaim smaller sections of memory after specific sets of instructions; for example, at the end of a loop iteration.

Real-time region types ensure threads do not use heap references, as execution may be interrupted by the garbage collector. Also, they cannot create new memory regions, or allocate objects in variable-time regions. Variable-time regions allocate memory on demand, and therefore are not as fast as the pre-allocated linear-time regions.

The type system is based on a series of relations to explicitly define region and object behaviour. For example, an ownership relation is used to define what owns an object. Objects can be owned by regions or other objects; every object must have an owner. This allows memory safety rules to be applied based on the relations, such as: the ownership relation has no cycles, and regions that own objects also own all subsequent objects owned by the object.

The outlives relation is used to define the order in which objects and regions can be reclaimed. For example, an object that owns another object must also outlive the same object because that is the only way it can be accessed. Similarly, if an object in a region r1 points to another object in a region r2, then the region r1 must outlive the region r2, otherwise a dangling reference could exist. This set of rules enforces the fact that no dangling references can exist. The dangling reference rules also apply to the subregions found in concurrent programs: objects allocated outside of the subregion cannot reference objects allocated inside the subregion.

The type system is enforced with a combination of type inference and defaults; the authors admit that annotating a program fully is an onerous task. A translation from their type system to standard RTSJ is presented; in reality, the aspects of the type system described match the components in RTSJ very well. The technique is presented as being suitable for a range of languages, however, the type system has been designed specifically with RTSJ in mind. The elimination of run-time checks has allowed the authors to produce RTSJ programs that run faster than normal implementations; they present a realistic speed-up figure of 1.25 times faster without the checks.

2.3.4 Dynamic logic

Engel presents a technique to statically verify RTSJ programs using the KeY system [15]. The KeY system facilitates the verification and specification of a sequential subset of Java using dynamic logic (JavaDL) [5]. The Java subset is JavaCard [13], a restricted language designed specifically for use on smart cards. Dynamic logic is a modal logic that gives a way to reason about states and programs; for example, given states s and s', and a program p, the formula $s \Rightarrow [p]s'$ is true if, and only if, the execution of p in every state satisfying s will either reach s' or not terminate.

Dynamic logic allows the addition of programs (a sequence of valid JavaCard statements) to be included in the specification; deduction uses symbolic program execution and simple program transformations.

The technique imposes certain restrictions on the RTSJ language and programming style in order to make verification easier. Specifically, the use of the heap is forbidden; the use of the heap is not recommended in hard real-time programs, therefore, the technique only caters for programs that exclusively use the immortal and scoped memory areas. These restrictions take the memory model of RTSJ closer to that of SCJ.

In order to specify the memory model of RTSJ in JavaDL, the semantics of the relevant RTSJ API components are described using a reference implementation and a series of JML [26] invariants. JML is a behavioural interface specification language for Java; it is used to express additional information about the interface and behaviour of Java programs using annotations. Interface properties describe the names and static information about Java declarations; behavioural properties describe how a declaration acts when called. The behavioural annotations are used to define pre and post conditions, and invariants, for example.

In addition, the RTSJ scope stack is described by a new abstract class with JML invariants. The nesting relation, which enables the technique to describe the order and relationship between scopes, is defined as a partial order. Finally, a set of calculus rules to describe the symbolic execution of RTSJ programs are defined. It is these rules that govern the memory model; for example, there exists a rule that ensures that for all reachable program states, the set of non-static attributes that are not null must obey the scope rules outlined above.

As the technique uses symbolic execution and program transformation, the calculus rules defined specify the laws of the memory model for every possible scenario. For example, consider an assignment to an object's field: `o.f = a`, which must accommodate the following possibilities when checking for null pointers:

- The object `o` is not null; therefore, the assignment is legal.
- The object `o` is null; therefore, a null pointer exception is raised.
- Neither of the two cases above can be established as true, and an illegal assignment error is raised.

When checking for violations in the RTSJ scoped region discipline, the rules specify that no dangling references can exist. More specifically, a reference variable must point to an object that resides in immortal memory, the same scoped memory area, or a scoped memory area higher in the structure, that is, towards the root. This is checked with an ordering on the scopes defined in a program and a record of the scopes in which objects reside.

Although this technique provides a formalisation of the RTSJ memory model, and a set of calculus rules to ensure programs obey the memory safety rules, it is limited by the restriction that the KeY system can only verify sequential programs.

2.3.5 Bytecode analysis

A popular method of analysing Java programs is through the bytecode, whose semantics is simpler than source code. Bytecode analysis includes many simplifications over the source code, including language-independence and name resolution, for example. The trade-off, however, is the lack of precision through the absence of high-level structures. Comparisons have been made between source code and bytecode analysers; these comparisons have shown that the relative completeness of bytecode analysis, in comparison to source code analysis, cannot always be guaranteed [27]. For example, source-code analysis is able to use the program structure of loops to increase precision. It does, however, make analysis more difficult, as analysing bytecode does not have to handle tasks performed by the compiler such as name resolution and type checking. As static analysis techniques are undecidable [25], the lack of precision found in bytecode analysis is often accepted as the benefit of easier analysis is preferred. It is, however, possible to develop static analysis techniques at the source code level that consider the worst-case scenario.

The technique in [43] is a context-sensitive, flow-sensitive points-to analysis based on Java bytecode. Specifically, the analysis uses an intermediate representation of the program that is generated from the bytecode and used by the JamaicaVM [1], which is a Java virtual machine designed specifically for hard real-time systems.

The intermediate representation generated by the JamaicaVM is more fine-grained in comparison to the bytecode; for example, an array access is split into four instructions to check the array is not null, obtain the length of the array, check the index value, and read the array element. By performing the points-to analysis on this finer-grained representation, the technique is able to check for null pointers, check that the region-based memory model of RTSJ is correct, and also calculate the worst-case memory allocation and worst-case stack use.

The correctness of the RTSJ memory model is checked by analysing context information about the allocation contexts for invocations and types. Allocation contexts are recorded when entering new memory areas; the recorded allocation context is the default context for runnable objects in that memory area.

The lack of scope cycles is checked using an ordering relation between contexts. Assignments are checked based on the allocation context of the target, and the allocation context of the invocation that is currently executing. If the allocation context of the target is not equal to, or a parent of, the invocation context, a potential error is raised.

This technique raises false-negatives, which are potential errors according to the analysis that can never be errors at run-time; it does not miss errors, however. In other words, the technique is sound, but not complete. An example of this is where a value may determine the execution path of a program, and potentially lead to an error on one path, but not another. As static analysis cannot determine which path will be executed, both options must be considered, therefore a false-negative is raised even if it is clear to the programmer that the error can never occur.

2.3.6 Conclusion

Techniques to verify the memory safety of RTSJ programs discussed here are focused on the removal of run-time checks to maximise the efficiency of the code. The overheads associated with run-time checks affect execution performance times, however the disadvantage is that removing them completely also removes the possibility to recover from any errors that may occur.

Many of the techniques discussed restrict the programming or memory model of RTSJ to facilitate checking of memory safety; these restrictions bring the paradigm closer to that of SCJ. Existing techniques to check memory safety of SCJ programs are discussed next.

2.4 Verifying memory safety in SCJ

As described previously, the SCJ language restricts the memory model of RTSJ by removing the heap. Memory areas in SCJ also follow a more rigid structure outlined by the SCJ programming paradigm. Techniques to establish the memory safety of SCJ programs are relatively new and varied; several different methods including code annotations, model checking, correctness by construction, and bytecode analysis are described below.

2.4.1 SCJ Annotations

Work to use static analysis in conjunction with SCJ annotations has been developed into a tool to check the conformity of SCJ programs against the language specification in [45]. The SCJChecker is designed to check the behavioural and level compliance annotations in the SCJ specification, but it also checks additional memory-safety annotations. The memory-safety annotations are the focus of this discussion, and are presented below.

- **@DefineScope** - This is used to explicitly define a new scope and takes two arguments: the name of the new scope and the parent scope.
- **@Scope** - This annotation restricts the scope in which a class, field, method, or local variable must reside.
- **@RunsIn** - This is used to define the scope in which methods execute, which may be different to the scope in which the enclosing class has been instantiated.

The static checking is achieved in two passes of the code. The first is used to produce an abstract syntax tree of the program; it also produces a scope stack based on the memory annotations and ensures no duplicates or cycles exist. Every scope stack must end with the immortal memory area. The second pass is used to actually check the program against the rules that accompany the annotations.

The memory-safety annotations are designed to impose restrictions on programs in order to prevent dangling references. The rules designed to ensure memory safety according to the SCJChecker are listed below.

- Objects must not be allocated outside the context defined.
- Arrays must not be allocated outside the context of their element type.
- Variables can only be declared in the same scope or those found further up the scope stack; that is, parent scopes cannot contain references to child scopes.
- Static variables must reside in the immortal scope, or have no annotations.
- Overridden methods inherit annotations from the super method.
- Method invocation is only allowed when the allocation context is the same, or the current context is a child of the method's allocation context.
- The `executeInAreaOf()` method can only be called on parent scopes; the runnable object passed as a parameter must be annotated to run in the corresponding scope.
- The `enterPrivateMemory()` method must be accompanied with an annotation on the runnable object that defines a new scope, where the new scope is a child of the current scope, and also has a RunsIn annotation defining its execution in the new scope.

- The `newInstance()` and `newArray()` methods can only be called if the element type is allowed (through annotations) to be allocated in the target scope.
- When casting a variable, the scope must be the same as the target type, or the type's scope must be undefined at which point the current allocation context must be the same as the current variable.

Rules also need to be defined for unannotated classes; the SCJChecker has the following inbuilt rules.

- Unannotated classes may be instantiated anywhere.
- Unannotated classes may not be passed as parameters outside the context in which they were instantiated.
- Methods that return unannotated objects must allocate them in their own context.
- Unannotated classes may not reference annotated objects.

All of these rules are checked to ensure that a given SCJ program is correct according to the constraints described by the annotations.

It is important to note that it is possible to write a correct SCJ program according to the language specification that is not considered correct based on the annotation constraints above. This is because the rules of the SCJChecker are stricter than those found in the specification; the rules are stricter in order to make the static analysis of the program easier.

As an example, consider the possible requirement that a class may be instantiated in different memory areas; by using the SCJ annotations, classes are restricted to a particular scope. To fulfil this requirement, it is necessary to duplicate the class in the code with a different `@Scope` annotation for each scope that may have instances of the class. Alternatively, the class can be defined once without a specific scope annotation; however, this becomes restricted by the rule that unannotated classes may not reference any other annotated objects.

The technique is also restrictive with respect to reference variable assignments across different scopes. For example, consider a class whose instances are defined to reside in the per-release memory area of a handler. According to the memory safety rules of SCJ, it is perfectly acceptable for an object that resides in the per-release memory area of a handler to reference an instance of the class in the mission memory; however, this raises an error in the checker to prevent the loss of scope knowledge.

The restrictions described here do not affect all SCJ programs; for example, the SCJChecker is more than capable of confirming that a given SCJ program that does not have a complex use of memory conforms to the SCJ specification. When the use of memory in a program becomes more complex, that is, when objects are passed between memory areas and classes are instantiated in different areas, the SCJChecker has limitations that mean potentially valid programs cannot be accepted, or require modifications. The SCJChecker is not limited to the memory properties we focus on here; it is also capable of checking behavioural and level properties of SCJ programs with annotations.

2.4.2 Model checking

Java Pathfinder (JPF) is a model-checking tool for Java programs designed to automatically analyse a multi-threaded Java program [18]. JPF uses a custom JVM that executes concurrent Java programs in every possible way, ensuring that every possible execution path is explored from a particular decision or instance of nondeterminism point (called choice points). This is achieved using a state graph to represent the choice points in the program execution.

Choice points occur when an input value is recorded or a particular thread is chosen for execution. JPF records information about the current state, and checks the previously visited states to ensure the same paths are not explored repeatedly. The state graph is used for backtracking to ensure every execution path from a particular choice point has been explored; considering all possible execution paths allows the model to be exhaustively checked. Symbolic model checking allows JPF to check all possible input values as opposed to a single value; this removes the need to produce tests to establish code coverage of all valid inputs.

As the state space of concurrent programs increases rapidly, JPF suffers from the state explosion problem. To tackle this, JPF uses an on-the-fly partial-order reduction technique, which combines operations that do not require communication between different threads into a single state. This is because the communications and interactions between threads in concurrent programs represent choice points in the execution path.

The $\mathsf{R_{SJ}}$ tool is an extension to JPF to support real-time Java programs, specifically RTSJ and SCJ programs [22]. To overcome the limitations of JPF described above, and make JPF suitable for real-time programs, a scheduling algorithm that implements fixed-priority preemptive scheduling without time-slicing is at the heart of R_{SJ}. The reason for using this algorithm is unclear, however, it is a common scheduling algorithm, widely used within the real-time sector.

The R_{SJ} algorithm is a more recent version of the older R_{J} algorithm [32]. The original R_{J} algorithm was designed for RTSJ, whereas the newer R_{SJ} is designed for both RTSJ and SCJ. One of the key differences between the two algorithms comes from the underlying platform used.

R_{J} is a platform-independent implementation that has no concept of code execution timings; this leads to a series of unrealistic schedulings. It was carried out without considering timing properties due to the unavailability of a precise timing model for execution [32]. It does, however, present a method to check time-independent RTSJ programs in JPF using a fixed-priority preemptive scheduling algorithm.

The newer R_{SJ} algorithm is based on the Java Optimized Processor (JOP), which is a hardware implementation of the JVM [39]. By using this platform optimised to give time-predictable results, R_{SJ} is a more efficient algorithm that explores fewer impossible schedulings.

The R_{SJ} algorithm is designed to operate with both RTSJ and SCJ programs; however, in reality only a subset of SCJ programs can be checked. More specifically, Level 0 and Level 1 programs are supported, however, there is no support for multi-processors or aperiodic event handlers (APEHs). Multi-processor programs, which are valid at Level 1, are not included as the authors suggested at the time of writing that certification of multi-processor Java applications seems not to be possible in the near future [22]. Their approach is therefore restricted to the analysis of Level 1 SCJ programs without aperiodic event handlers on single-core systems. APEHs are ignored also due to the inevitable state explosion that occurs if included. Due to the unknown release times of aperiodic events, including these in the checking process would effectively require the algorithm to nondeterministically decide whether to release a currently non-released APEH after every instruction.

In SCJ, APEHs are realistically sporadic as opposed to true aperiodic handlers; therefore, it would be possible to assume the worst-case scenario where the APEH was released at every possible point after the minimum inter-arrival time. This does not, however, get over the problem of actual arrival times, where the release of an APEH may not coincide nicely with the minimum inter-arrival time; by using a model checking technique, it would still be essential to check all possible arrival times.

Level 0 SCJ programs contain no preemption or concurrency, and hence have only one possible scheduling. As R_{SJ} does not make use of JPF's symbolic execution mode, it cannot discover any additional errors to those found through testing at run-time, without the addition of assertions in the original code. At Level 1, more errors can be found than standard testing as every possible scheduling is executed. When focusing on memory safety, the R_{SJ} tool can be used to detect memory-access errors, dereferencing of null pointers, and array-bound violations. It is also capable of checking the memory structure of SCJ programs to ensure they match the rules outlined in the SCJ specification. This is achieved by simply checking for exceptions that may arise during the model checking process. If an exception occurs from a dangling pointer error then the assignment to the reference variable breaks the memory safety rules of SCJ.

As R_{SJ} is able to detect unhandled exceptions and report these as errors to the user, it is also possible to check memory properties to ensure RTSJ or SCJ implementations are memory safe. Null-pointer exceptions are found and reported like in JPF. Violations of the memory structure produce exceptions, for example, when a reference in immortal memory points to an object in mission memory.

R_{SJ} operates on top of JPF, which is designed for use on standard Java implementations; therefore, the memory model of RTSJ and SCJ is modelled on top of Java. Figure 2.11 shows how immortal and scoped memory areas are used in conjunction with the standard Java programming model. The RTSJ scope stack on the left is used to store reference and primitive values of variables in the individual scopes shown. Each new scope contains a reference to the current memory area; for example, the variable ma in the first scope points to the immortal memory area ima. Scoped memory areas are represented as objects placed on the heap; this is a way of implementing the scoped memory model on top of JPF. Similarly, objects that are allocated to a specific memory area are placed on the heap like any other object; a reference to the memory area in which it lies is stored as an additional field in the object.

Memory properties are managed in R_{SJ} using a JPF listener that monitors bytecode execution; the listener is used to maintain the program stack and memory areas. Garbage collection is disabled for memory area objects to maintain the properties of scoped and immortal memory. References to memory areas are not stored on the heap to ensure they

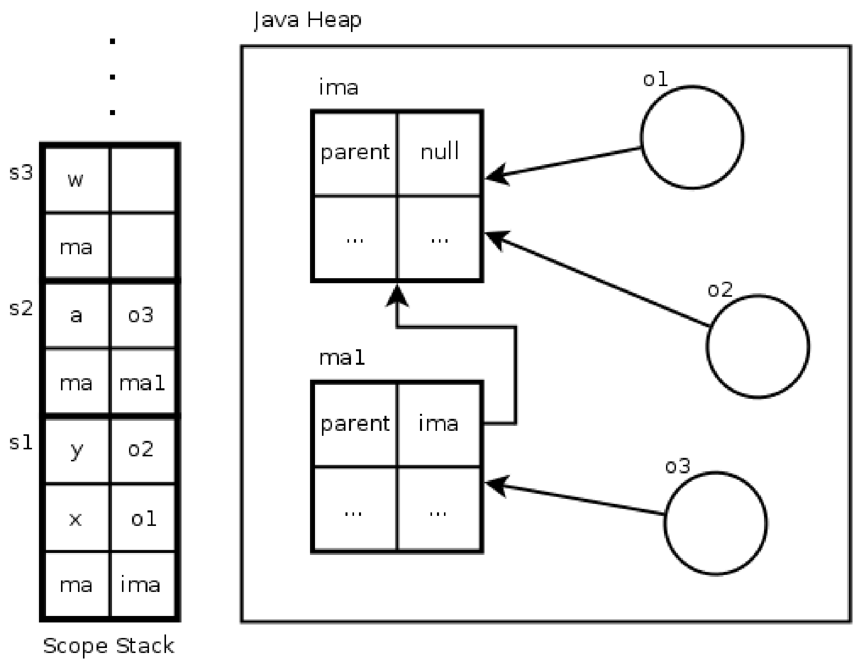


Figure 2.11: RTSJ and SCJ memory representation in R_{SJ}

are not removed by the garbage collector; instead, they are stored as a field in each scope on the stack, as described above.

The lack of support for concurrency or APEHs makes it difficult to justify the author's claim of support for Level 1 programs. Model checking concurrent programs is a difficult problem that quickly encounters issues with the state explosion problem. To fully take advantage of the technique presented here, the symbolic execution offered by JPF must be utilised to ensure that every possible set of input values still produce a memory safe Level 0 program; however, this would not overcome the problems that limit this technique to sequential programs.

2.4.3 Correctness by construction

An alternative to checking implementations are memory safe is to use a correctness-by-construction technique to ensure the resulting implementation is guaranteed to be memory safe. Cavalcanti *et al.* present a technique in [12] that uses the formal language *Circus* [51] to define abstract models; these abstract models are used in a series of refinement stages for the development of correct Level 1 SCJ programs.

The *Circus* language is based on a combination of Z [52], CSP [36], and Dijkstra's guarded commands [29]. *Circus* specifications are made up of a series of processes; Z schemas describe the functional behaviour, whilst CSP is used to model the communication and execution order of processes. The semantics of *Circus* is defined in the UTP [20], which has existing theories about object-orientation and time; both of which are relevant to SCJ.

In order to fully specify the SCJ paradigm, additional *Circus* variants including *CircusTime* [42], which includes features of Timed CSP [34], and *OhCircus* [10] are used to express timing properties and object-oriented features. Timed CSP allows specifications to describe timing properties such as wait and timeout. *OhCircus* introduces classes; behaviour is described through methods as opposed to actions.

The authors have defined a formalisation of both the SCJ mission model [53] and the SCJ memory model [11] which in conjunction with the semantics of *Circus* and the UTP gives them a solid verification platform.

By using the UTP, it allows the memory model to be integrated with existing theories about object-orientation and time, both of which are relevant to SCJ. The memory model has associated healthiness conditions that present a foundation to allow formal reasoning about memory safety of SCJ programs. A program that conforms to these conditions is guaranteed to be memory safe. This is currently the only formal definition of the SCJ memory model. Sound approaches to verifying memory properties can be investigated using this model.

The refinement strategy is split into four parts, characterised by models that follow particular architectural patterns, and are called anchors, which define development steps from a high-level specification to a low-level one that facilities code generation; each anchor is a refinement of the previous one.

The first anchor, or the A anchor, represents the abstract model of the program. At this level, there is no information about the structure of an SCJ program, classes, or objects; instead, only the interaction of the system is described. Parallelism is used in this anchor to represent the combination of multiple requirements, as opposed to the concurrency of an implementation.

The second anchor, called the M anchor, introduces the concept of memory allocation; classes and objects are also included. Data refinement is used to replace Z data types with class types; these class types represent references to objects, whilst Z types represent values.

The third anchor is the E anchor; this describes the execution model of the SCJ paradigm; it is at this refinement stage that the concept of missions and handlers are introduced into the model. This refinement step is itself split into four stages:

1. The parallel composition of requirements in the A anchor are removed, as these are only useful for the specification of requirements, and not the definition of the SCJ paradigm.

2. The location of objects and shared data is based on the SCJ memory structure.

3. Actions are defined for each mission and handler; the corresponding sequence of missions and parallelism of handlers is defined.

4. Algorithmic refinement is used to define the implementation of individual methods and handler routines.

Finally, the fourth anchor, or the S anchor, is used to describe the E anchor in terms of the SCJ framework. The overall specification is the parallel composition of the safelet, mission sequencer, missions, and handlers. Each of these components is also a parallel composition of two processes that describe the generic behaviour found in the SCJ framework, and the actual behaviour of the specific program.

It is the S anchor that is very close to an actual SCJ implementation; and can be used for code generation. Similarly, it is possible to reverse the translation and produce models in the S anchor from SCJ programs if the SCJ program follows a specific design pattern. The authors require that, for example, programs have separate classes to define the safelet, mission sequencer, missions, and handlers. Whilst this is not an unreasonable expectation, as it follows the structure presented in the SCJ specification, it is possible to produce cyclic executive SCJ programs that only use a single class, which would not conform to the requirements.

The advantage of generating S models from implementations comes from the memory assurances automatically gained, assuming the implementation can be expressed successfully. Using correctness by construction, and the underlying formal representation of the technique, it is guaranteed that any implementation that can be successfully expressed as an S model is memory safe.

The main challenge that remains for the authors is the proof of soundness required to backup the technique. A more in-depth description of the technique, and a real-world case-study to demonstrate its applicability are shown in [54]. The overall development technique is an idealised approach that requires skilled users to take full advantage of its potential; however, the possible automatic production of S models from implementations to guarantee memory safety is appealing.

2.4.4 Bytecode analysis

As mentioned previously, bytecode analysis is a popular and simpler analysis technique for Java programs in comparison to source code analysis. Dalsgaard *et al.* have developed a context-sensitive points-to analysis technique for SCJ bytecode that checks for possible memory-safety violations according to the SCJ specification [14].

The analysis is based on WALA [4], which supports pointer analysis of Java bytecode in user-defined contexts. WALA creates an intermediate representation of the bytecode in static single assignment (SSA) form for the analysis.

The points-to analysis gives an over-approximation of the possible memory-safety violations in a program by recording all possible references between variables and objects. A stack of SCJ memory areas gives context to the references and the scopes in which associated objects reside, making checking for potential violations possible.

The definition of a scope (in this work) in which each object is allocated has two components: the memory area identifier, and the type of scope. The memory area identifier is a unique identifier associated with the specific memory area, whilst the type of scope is defined as either immortal, mission, private memory, or unknown. The unknown scope type is used when it is not possible to determine the particular scope of an allocation. An ordering is defined on the possible scopes of a program, which is used as the basis for checking whether a reference from one scope to another is valid or not.

The authors identify some of the SCJ-specific methods mentioned in Section 2.2 that may have a direct impact on memory safety; these are tracked throughout the analysis using a call graph. Other methods such as `newArray`, `newInstance`, `getMemoryArea`, and `executeInAreaOf` are currently not handled.

The SCJ implementation used for the analysis is based on JOP [41]; possible errors raised by the SCJ implementation are filtered out of the results as the authors state that it is acceptable for the SCJ infrastructure to temporarily break the scope rules. This does not conform to the definition of memory safety previously presented in Section 2.2, which states that any downward reference is a possible memory-safety violation. Without an official reference implementation for SCJ, it is difficult to see why the infrastructure would need to break the rules of the memory model.

The reference implementation of SCJ is yet to be published, and decisions on how to implement the memory structure are still undecided. The review in [40] describes the two main possibilities of either using the existing RTSJ library and extending it for SCJ, or building a new representation in the SCJ library.

The authors also describe techniques to use additional commands found in the JOP/SCJ implementation to successfully track the execution of SCJ-specific commands, such as the `startMission` method. These additional commands, which are not part of the SCJ specification, are only required to track the execution of the program because the analysis is at the bytecode level as opposed to the source code level. The technique suffers from unnecessary false negatives that are raised by problems tracking the start of missions.

A number of optimisations have been applied to the analysis technique based on the JOP/SCJ implementation to reduce the number of false negatives raised during the analysis. Specifically, a number of objects allocated by the JOP implementation in the initialisation phase are ignored. It is apparent that the analysis is not completely general, and has been tailored to the JOP/SCJ implementation to improve its performance.

By using an intermediate representation of the bytecode, mapping possible errors back to the source code is more difficult. Currently potential errors are reported based on the names of fields, methods, and classes, which the authors have found sufficient.

The technique is sound but not complete, as false-negatives may be raised; however, the soundness of the technique has not been proved. Overall, the technique demonstrates that it is possible to produce a static checking technique that identifies possible memory-safety violations in SCJ programs, without the need for additional user-added annotations.

2.4.5 Hardware checking

Memory-safety violations that occur at run-time raise exceptions; techniques described above are aimed at removing run-time exceptions through static analysis, program restriction, or annotations. An alternative to this is to implement the checking technique in hardware, which does not suffer from the same overheads as software-based approaches.

The technique in [35] describes a hardware checking technique, based on JOP, for memory-safety violations in SCJ. Not only is the software-checking overhead eliminated, but the execution time of applications with large numbers of cross-scope references is improved. The trade-off with this approach is that it is hardware specific, and cannot be applied at the more general bytecode or source-code level.

Reference assignments are checked at the hardware level using write barriers [19], which are additional pieces of code placed at every instruction that may impact object references. The additional code checks the scopes of the two objects to ensure that the assignment will not cause a memory-safety exception. The scope level of each object is stored in the corresponding reference pointer, which makes the checks possible through simple arithmetic.

This method of checking bytecodes at the hardware level is about ten times faster than software versions, and does not have a significant impact in terms of additional memory required to store the required scope information. Also, when applied to several benchmarks, the execution time decreased by as much as 18%, thus demonstrating the possible speed-up achieved through removing software run-time checks.

2.4.6 Conclusion

Techniques to verify the memory safety of SCJ programs vary, as indicated by their description above. Currently, the most complete approach to checking memory safety of SCJ implementations comes from the SCJ bytecode analysis technique; however, bytecode analysis is simpler than source code analysis, and can give more false-negatives based on the difficulty to track execution through the SCJ paradigm.

The other comprehensive approach is the SCJChecker, which relies on the SCJ annotations; however, the strict restrictions imposed on programs make its practical use questionable. The formal outline of the SCJ memory model defined using the UTP in [11] gives a basis on which a sound checking technique can be based, however, currently no such technique exists.

The use of model checking in safety-critical systems is widely used, and efforts are still being undertaken to apply them to SCJ. The state explosion problem will remain a problem in concurrent systems and therefore other static checking techniques are being investigated.

2.5 Summary

This chapter has introduced the real-time and safety-critical variants of Java, and has described the different memory models for each. The use of region-based memory models has allowed the development of real-time and safety-critical software in the Java language. Investigation into real-time garbage collection is still ongoing, however, as program development with a region-based memory model is more difficult.

The programming paradigm and memory model defined in the RTSJ specification has been restricted in several techniques, thus prompting the development of SCJ, an even tighter language designed specifically to aid static verification and to ease certification. Several techniques to verify memory safety of SCJ programs have been presented, however, there is currently no practical solution at the source-code level for Level 1 SCJ programs that does not impose unwanted restrictions on program development.

The next chapter presents the outline of a source-code level static checking technique that tries to limit the restrictions imposed by other techniques. In particular, the use of annotations is avoided, and there are no further programming restrictions other than those outlined in the SCJ specification.

The static checking technique presented in the next chapter is backed up with a mathematical formalisation of all aspects of the analysis, which gives a solid foundation in order to prove the soundness of the technique. More specifically, it gives a precise definition of the intermediate representation used, the translation strategy from SCJ to the intermediate representation, and the checking technique used to detect potential memory-safety violations.

Chapter 3

SCJ-mSafe: An abstract language for memory-safety checking

In this chapter, the *SCJ-mSafe* language is introduced; it is an intermediary language used in the verification technique for SCJ programs. Section 3.1 demonstrates how the SCJ program presented in the last chapter is defined in *SCJ-mSafe*. Section 3.2 defines a formal model of SCJ programs in Z. Section 3.3 defines a formal model of *SCJ-mSafe* programs. Together these two language models can be used to define a translation. An overview of the translation is presented in Section 3.4 before a formal translation strategy in Z is defined in Section 3.5. Finally, Section 3.6 draws some conclusions.

3.1 An example program in *SCJ-mSafe*

This section demonstrates how a case study implemented in SCJ is defined in *SCJ-mSafe*. The *SCJ-mSafe* translation of the cruise controller example presented in the previous chapter is shown in Figures 3.1 and 3.2.

Figure 3.1 shows the same extract from the `ACCMission` class in Figure 2.5 in *SCJ-mSafe*. The `initialize` method in the SCJ implementation is represented as a specific component of the mission in *SCJ-mSafe*, as opposed to a normal class method definition, because it is an integral part of the SCJ paradigm. The first three simple method calls remain unchanged. The creation of the handlers are now separated into two individual commands for each one: a declaration, and an instantiation. Consider the creation of the `WheelShaft` handler, which is first declared as the uninstantiated variable `shaft` (line 10, Fig 3.1). This is then instantiated on the next line (line 11, Fig 3.1) with the `NewInstance`

```
mission ACCMission {

  ...

  initialize {
    createEvents();
    createISRs();
    registerISRs();

    WheelShaft shaft;
    NewInstance(shaft, Current, WheelShaft, (shaft_event));
    SpeedMonitor speedo;
    NewInstance(speedo, Current, SpeedMonitor, (shaft, Val));
    ThrottleController throttle;
    NewInstance(throttle, Current, ThrottleController,
       (speedo));
    Controller cruise;
    NewInstance(cruise, Current, Controller, (throttle,
       speedo));
    Engine engine;
    NewInstance(engine, Current, Engine, (cruise,
       engine_event));
    Brake brake;
    NewInstance(brake, Current, Brake, (cruise,
       brake_event));
    Gear gear;
    NewInstance(gear, Current, Gear, (cruise, gear_event));
    Lever lever;
    NewInstance(lever, Current, Lever, (cruise,
       lever_event));

    shaft.register();
    engine.register();
    brake.register();
    gear.register();
    lever.register();
    speedo.register();
    throttle.register();
  }

  ...
}
```

Figure 3.1: ACCS Mission in *SCJ-mSafe*

command, which states that the object associated with the variable `shaft` is instantiated in the default context, with type `WheelShaft`, and that the corresponding constructor that accepts an `AperiodicEvent` as its argument (`shaft_event`) is called. The remaining handlers and classes are declared and instantiated in the same way.

Figure 3.2 shows the `SpeedMonitor` handler from Figure 2.6 in *SCJ-mSafe*. Similarly to the creation of the handlers in the mission class, the fields of the handler are also separated into two parts: the declaration, and the instantiation. Instead of placing the instantiation directly after the declaration, field declarations and instantiations are separated completely into two components called `fields` and `init`, respectively. This is because it is useful to know the fields of a class during the analysis, and this can be modelled better as a set of fields rather than a mix of declarations and instantiations.

Most of the fields in the `SpeedMonitor` are of primitive type (line 2, Fig 3.2); their initialisation commands are abstracted away. It does not matter what calculations are performed to determine the result, as it will always be a value (as opposed to a reference), and values are not important in memory-safety analysis. The translated commands are shown on line 11 of Figure 3.2.

The first statement in the constructor of the handler in SCJ is a call to the method `super` (line 12, Fig 2.6), however its arguments contain several embedded method calls and instantiations. In *SCJ-mSafe*, these are extracted out as individual commands; as such, new variables are introduced to record the results of method calls and instantiations for use later in the program (lines 17-28, Fig 3.2). The first three statements in the constructor in *SCJ-mSafe* are declarations; these are all new variables required to construct the `PriorityParameters` object that specifies the priority of the handler.

The fourth statement, which is a call to the `PriorityScheduler.instance` method (line 20, Fig 3.2) takes `var11` as an argument. In the SCJ version (line 24, Fig 2.6), this method was parameterless, however in *SCJ-mSafe*, because the method returns an object, and it is now a command as opposed to an expression, an additional result variable is required. In this example, the result variable is `var11`. The method `getMaxPriority` is then called on the new variable `var11` (line 21, Fig 3.2), as each method call is separated out into a single command; the result of this is stored in the variable `var12`. Finally, `var10` is instantiated with a new instance of the `PriorityParameters` class, which takes the max priority (`var12`) as a parameter (line 22, Fig 3.2).

The remaining statements are separated into their individual parts, with additional variables being introduced to maintain references to values and objects where necessary. This process has made the *SCJ-mSafe* version of the code longer in comparison to the SCJ code, however, this makes the definition of a formal checking technique much easier, as all the possibilities of nested commands and expressions do not need to be considered.

The local method `getCurrentSpeed` demonstrates how the result parameter is defined and used (line 48, Fig 3.2). Whereas previously, in SCJ, the definition of the method

```
handler SpeedMonitor {
  fields {
    int calibration;
    int iterationsInOneHour;
    int cmInKilometer;
    WheelShaft wheel_shaft;
    long numberRotations;
    long lastNumberRotations;
    int currentSpeed;
  }
  init {
    cmInKilometer = Val;
    numberRotations = Val;
    currentSpeed = Val;
  }
  constr (shaft, period) {
    PriorityParameters var10;
    int var12;
    PriorityScheduler var11;
    PriorityScheduler.instance(var11);
    var11.getMaxPriority(var12);
    NewInstance(var10, Current, PriorityParameters, (var12));
    PeriodicParameters var13;
    RelativeTime var14;
    NewInstance(var14, Current, RelativeTime, (period, Val));
    NewInstance(var13, Current, PeriodicParameters, (null,
       var14));
    StorageParameters var15;
    NewInstance(var15, Current, StorageParameters, (Val,
       Val, Val));
    super(var10, var13, var15, Val);
    int var16;
    wheel_shaft.getCallibration(var16);
    calibration = var16;
    iterationsInOneHour = Val;
    wheel_shaft = shaft;
    long var17;
    wheel_shaft.getCount(var17);
    lastNumberRotations = var17;
  }
  handleEvent {
    long var18;
    wheel_shaft.getCount(var18);
    numberRotations = var18;
    long difference;
    difference = Val;
    currentSpeed = Val;
    lastNumberRotations = numberRotations;
  }
  method getCurrentSpeed(Result) {
    Result = currentSpeed;
  } }
```

Figure 3.2: SpeedMonitor Handler in *SCJ-mSafe*

included a `return` statement (line 35, Fig 2.6), the *SCJ-mSafe* version performs an assignment from the variable `currentSpeed` to the new result parameter `Result` (line 49, Fig 3.2). Any expression passed as an argument to the `getCurrentSpeed` method will be assigned the value stored in `currentSpeed`.

The translation from SCJ to *SCJ-mSafe* is described in more detail in Sections 3.4 and 3.5.

3.2 A formal model of SCJ

In order to define a formalisation of the translation approach, both the SCJ and *SCJ-mSafe* languages have been formalised in Z. This allows us to define the translation, analysis, and checking strategies later. A description of the necessary components of the SCJ formalisation is presented in this section; the full version can be found in Appendix B. A table showing a description of the Z notation used can be found in Appendix A.

The models of each language define a set that contains as elements abstract representations of the terms of the language. At the top level of the model is a definition of an *SCJProgram*, which is a description of the overall SCJ program being analysed. Inside each *SCJProgram* is a set of classes defined by the *SCJClass* schema, which are in turn made up of a set of fields and methods. At the lowest level of the model are the commands and expressions, which are defined by the types *SCJCommand* and *SCJExpression* respectively.

The top level of the model of an SCJ program is represented by the *SCJProgram* schema, which has a single component *classes* recording the set of classes of the program.

SCJProgram

$classes : \mathbb{P}\ SCJClass$

There is no distinction between the specific components of the SCJ paradigm, such as missions or handlers, at this level. Every class inside the SCJ program is represented as an *SCJClass*, which is made up of class modifiers, a name, type parameters, an extends class, an implements class, and a sequence of class components.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;SCJClass\;}}\\
modifiers : SCJModifier\\
name : Name\\
typeParameters : \mathbb{P}\ TypeParameter\\
extends : Name\\
implements : Name\\
members : \mathrm{seq}\ SCJClassComponent\\
\hline
\end{array}
$$

The modifiers of the *SCJClass* define whether the class is public, abstract, or final, for example; the *SCJModifier* type represents either a Java flag or annotation, but is omitted here. The class name is defined to be of type *Name*, and the type parameters used for generic definitions are defined as a set of *TypeParameter*s. The extends component is of type *Name* and describes the extended class if applicable. The types *Name* and *Value* represent names and values respectively; the model of the language includes some explicit definitions of names in order to facilitate the identification of specific components. The implements clause is also of type *Name* and describes the class being implemented. Finally, the class components, which are fields and methods of the class, are represented as a sequence of type *SCJClassComponent*.

$$SCJClassComponent ::= ClassField\langle\langle SCJVariable\rangle\rangle \mid ClassMethod\langle\langle SCJMethod\rangle\rangle$$

A class component is defined as either a variable, which is a class field represented by the *SCJVariable* schema, or a class method, which is represented by the *SCJMethod* schema.

Variables in SCJ consist of four elements: modifiers, the type, the name, and an initialisation expression, which may be empty if the variable is declared but not initialised.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;SCJVariable\;}}\\
mods : SCJModifier\\
type : TypeElement\\
name : Name\\
init : SCJExpression\\
\hline
\end{array}
$$

Variable modifiers define whether the variable is public, private, or static, for example. The type of the variable is defined by the *TypeElement* schema. The name of the variable is defined to be of type *Name*, and finally the initialisation expression is defined to be an *SCJExpression*, which will be described later.

The other possible class component is a method, which is made up of the method modifiers, type parameters, a return type, a name, parameters, and the method body.

$$
\begin{aligned}
& \mathit{SCJCommand} ::= \\
& \quad \mathit{assert} \langle\langle \mathit{SCJExpression} \times \mathit{SCJExpression} \rangle\rangle \\
& \quad \mid \mathit{block} \langle\langle \mathit{Boolean} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{break} \langle\langle \mathit{Name} \rangle\rangle \\
& \quad \mid \mathit{continue} \langle\langle \mathit{Name} \rangle\rangle \\
& \quad \mid \mathit{doWhile} \langle\langle \mathit{SCJExpression} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{empty} \\
& \quad \mid \mathit{eFor} \langle\langle \mathit{SCJModifier} \times \mathit{TypeElement} \times \mathit{Name} \times \mathit{SCJCommand} \\
& \qquad \times\ \mathit{SCJExpression} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{expression} \langle\langle \mathit{SCJExpression} \rangle\rangle \\
& \quad \mid \mathit{for} \langle\langle \mathit{SCJCommand} \times \mathit{SCJExpression} \times \mathit{SCJCommand} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{if} \langle\langle \mathit{SCJExpression} \times \mathit{SCJCommand} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{labeled} \langle\langle \mathit{Name} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{return} \langle\langle \mathit{SCJExpression} \rangle\rangle \\
& \quad \mid \mathit{switch} \langle\langle \mathit{SCJModifier} \times \mathit{TypeElement} \times \mathit{Name} \times \mathit{SCJCommand} \\
& \qquad \times\ \mathit{SCJExpression} \times \operatorname{seq} \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{synchronized} \langle\langle \mathit{SCJExpression} \times \mathit{Boolean} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{throw} \langle\langle \mathit{SCJExpression} \rangle\rangle \\
& \quad \mid \mathit{try} \langle\langle \mathit{SCJCommand} \times \mathit{SCJExpression} \times \mathit{SCJCommand} \rangle\rangle \\
& \quad \mid \mathit{variable} \langle\langle \mathit{SCJVariable} \rangle\rangle \\
& \quad \mid \mathit{while} \langle\langle \mathit{SCJExpression} \times \mathit{SCJCommand} \rangle\rangle
\end{aligned}
$$

Figure 3.3: SCJ Commands in Z

$$
\begin{aligned}
& \underline{\mathit{SCJMethod}\qquad\qquad\qquad} \\
& \mathit{modifiers} : \mathit{SCJModifier} \\
& \mathit{typeParameters} : \mathbb{P}\ \mathit{TypeParameter} \\
& \mathit{returnType} : \mathit{TypeElement} \\
& \mathit{name} : \mathit{Name} \\
& \mathit{params} : \operatorname{seq} \mathit{SCJVariable} \\
& \mathit{body} : \mathit{SCJCommand}
\end{aligned}
$$

The method modifiers define whether it is a public, private, or static method, for example. The set of type parameters defined by the *TypeParameter* type are used for generic methods. The return type of the method is defined by a *TypeElement*. The method name is defined by the type *Name*, and the parameters of the method are defined as a sequence of variables. Finally, the body of the method is an SCJ command, which is defined by the *SCJCommand* type.

The definition of the type *SCJCommand* is shown in Figure 3.3. Commands are

$$
\begin{array}{l}
SCJExpression ::= \\
\qquad null \\
\qquad \mid annotation \\
\qquad \mid arrayAccess \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
\qquad \mid assignment \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
\qquad \mid binary \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
\qquad \mid compoundAssignment \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
\qquad \mid conditional \langle\langle SCJExpression \times SCJExpression \times SCJExpression \rangle\rangle \\
\qquad \mid erroneous \\
\qquad \mid identifier \langle\langle Name \rangle\rangle \\
\qquad \mid instanceOf \langle\langle SCJExpression \times TypeElement \rangle\rangle \\
\qquad \mid literal \langle\langle Value \rangle\rangle \\
\qquad \mid memberSelect \langle\langle SCJExpression \times Name \rangle\rangle \\
\qquad \mid methodInvocation \langle\langle SCJExpression \times \mathrm{seq}\ SCJExpression \rangle\rangle \\
\qquad \mid newArray \langle\langle TypeElement \times \mathrm{seq}\ SCJExpression \rangle\rangle \\
\qquad \mid newClass \langle\langle Name \times \mathrm{seq}\ SCJExpression \rangle\rangle \\
\qquad \mid parenthesized \langle\langle SCJExpression \rangle\rangle \\
\qquad \mid typeCast \langle\langle TypeElement \times SCJExpression \rangle\rangle \\
\qquad \mid unary \langle\langle SCJExpression \rangle\rangle
\end{array}
$$

Figure 3.4: SCJ Expressions in Z

made up from additional commands and expressions; for example, the *for* command is made up from three commands and one expression. These represent the loop initialisation command, the expression that determines whether or not to enter the body of the loop, the body of the loop itself, and finally the command executed after each iteration of the body.

SCJ expressions are represented by the type *SCJExpression*, which is defined in Figure 3.4; most expressions are made up from additional *SCJExpression*s. For example, the assignment expression has two additional expressions that represent the left and right-hand sides of the assignment.

This model of the SCJ language facilitates the definition of a formal translation strategy from SCJ programs to *SCJ-mSafe* programs; the formalisation of the *SCJ-mSafe* language is presented next.

3.3 A formal model of *SCJ-mSafe*

The formalisation of the *SCJ-mSafe* language in Z is presented in this section. The model is based on the extraction of the necessary information in the original SCJ program to check for possible memory-safety violations. The key components of the model are presented here, however, the full version can be found in Appendix C. All of the Java language features are handled except for lambda expressions. The use of generics is a typing issue that is checked by the Java compiler, and it does not affect the ability to analyse programs for potential memory-safety violations.

The *SCJ-mSafe* BNF is shown in Figures 3.5 and 3.6. Figure 3.5 shows the syntax for the overall program, safelet, mission sequencer, missions, and handlers. Figure 3.6 shows the syntax for individual classes, methods, commands, and expressions. The formalisation presented here is a model of the syntactic program that subsequently facilitates the definition of a formal translation strategy between SCJ and *SCJ-mSafe*.

3.3.1 *SCJ-mSafe* - Overall Program

An *SCJ-mSafe* program is defined by a set of static variables and their corresponding initialisations, a safelet, a mission sequencer, any number of missions, handlers, and user-defined classes. This is shown on line 1 of Figure 3.5.

Field declarations that are instantiated at the point of declaration are split into two separate commands, the first is the introduction of the new variable, and the second is the initialisation command that may create a new object, or assign a value to the newly declared variable, for example. The introduction of the new variable is defined as a declaration in the `fields` component, whilst the corresponding initialisation commands are recorded in the `init` component. Separating the declaration from the instantiation of a variable allows for a simpler analysis later, which uses information about the fields of a class without their corresponding initialisation commands.

The corresponding Z definition of the overall program is the schema *SCJmSafeProgram*, which is shown below.

```
    <Program> ::= static (<Declaration>*)
                  sInit  (<Com>*)
                  <Safelet>
                  <MissionSequencer>
                  <Mission>*
                  <Handler>*
                  <Class>*

    <Safelet> ::= safelet Name {
                      fields (<Declaration>*)
                      init   (<Com>*)
                      constr (<Method>)*
                      initializeApplication (<Com>)
                      getSequencer (<Com>)
                      method (<Method>)*
                  }

    <MissionSequencer> ::= missionSeq Name {
                               fields (<Declaration>*)
                               init   (<Com>*)
                               constr (<Method>)*
                               getNextMission (<Com>)
                               method (<Method>)*
                           }

    <Mission> ::= mission Name {
                      fields (<Declaration>*)
                      init   (<Com>*)
                      constr (<Method>)*
                      initialize (<Com>)
                      cleanUp (<Com>)
                      method  (<Method>)*
                  }

    <Handler> ::= handler Name {
                      fields (<Declaration>*)
                      init   (<Com>*)
                      constr (<Method>)*
                      handleEvent (<Com>)
                      method (<Method>)*
                  }
```

Figure 3.5: *SCJ-mSafe* BNF 1

SCJmSafeProgram

$static : \mathbb{P}\ Dec$
$sInit : \mathbb{P}\ Com$
$safelet : Safelet$
$missionSeq : MissionSeq$
$missions : \mathbb{P}\ Mission$
$handlers : \mathbb{P}\ Handler$
$classes : \mathbb{P}\ Class$

```
<Class> ::= class Name extends Name {
                fields (<Declaration>*)
                init   (<Com>*)
                constr (<Method>)*
                method (<Method>)*
            }

<Method> ::= method Name (variable*) {<Com>}

<Com> ::= skip |
          <LExpr> = <Expr> |
          NewInstance (<LExpr>, <MetaRefCon>, <VarType>,
             <Expr>*) |
          <Com> ; <Com> |
          <Declaration> |
          do {<Com>} while (<Expr>) |
          <LExpr>.Name (<Expr>*) |
          ExecuteInAreaOf (<Expr>, <Expr>) |
          ExecuteInOuterArea (<Expr>) |
          EnterPrivateMemory (<Com>) |
          GetMemoryArea(<Expr>, <Expr>) |
          for (<Com>; <Expr>; <Com>) {<Com>} |
          if (<Expr>) {<Com>} else {<Com>} |
          switch (<Expr>) {<Com>+} |
          try {<Com>} (catch (<Expr>) {<Com>})+ finally
             {<Com>} |
          while (<Expr>) {<Com>}

<Declaration> ::= Type Name

<Expr> ::= value | <Identifier> | <FieldAccess> |
   OtherExpr | null | this

<LExpr> ::= <Identifier> | <FieldAccess>

<Identifier> ::= variable | arrayElement[value]

<FieldAccess> ::= <Identifier>.<Identifier>+
```

Figure 3.6: *SCJ-mSafe* BNF 2

As defined in the BNF, the definition above includes the set of static variables (*static*), which is a set of declarations (*Dec*), the static fields corresponding initialisation commands (*sInit*), which is a set of commands (*Com*), a safelet (*safelet*) of type *Safelet*, a mission sequencer (*missionSeq*) of type *MissionSeq*, a set of missions (*missions*) of type *Mission*, a set of handlers (*handlers*) of type *Handler*, and a set of user-defined classes (*classes*) of type *Class*.

By defining *SCJ-mSafe* programs in this consistent structure, with all of the information required to check memory safety, it is possible to produce a checking technique that can be applied to any *SCJ-mSafe* program.

3.3.2 *SCJ-mSafe* - Safelet

An *SCJ-mSafe* safelet is the starting point of a program. It sets up the program through the `initializeApplication` method, which is the first method to be called by the infrastructure, and creates a mission sequencer in the `getSequencer` method.

As shown on line 9 of Figure 3.5, the safelet definition has the basic class components, which are a name, fields, field initialisation commands, constructors, and other user-defined class methods, along with an `initializeApplication` and `getSequencer` method.

The `initializeApplication` and `getSequencer` methods are defined as specific components of the safelet, as opposed to simply being one of the regular class methods, because their execution is a fundamental part of the SCJ paradigm. As they can only be called by the infrastructure, and the order in which the infrastructure calls these methods is known, the body of the method call can be extracted and defined as a command; there is no need to represent infrastructure-called methods as methods. The corresponding formalisation of the safelet class is shown below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Safelet}} \\
\hline
name : Name \\
fields : \operatorname{seq} Dec \\
init : \operatorname{seq} Com \\
constrs : \mathbb{P}\, Method \\
initializeApplication : Com \\
getSequencer : Com \\
missionSeq : MissionSeq \\
methods : \mathbb{P}\, Method \\
\hline
\end{array}
$$

The *Safelet* schema includes the safelet's name (*name*), which is of type *Name*, the class fields (*fields*), which is a sequence of declarations (*Dec*), the corresponding field initialisation commands (*init*), which is a sequence of commands (*Com*), the class constructors (*constrs*), which is a set of type *Method*, the `initializeApplication` method (*initializeApplication*), which is a command (*Com*), the `getSequencer` method (*getSequencer*), which is a command, the mission sequencer (*missionSeq*), which is of type *MissionSeq*, and finally the user-defined class methods (*methods*), which is a set of type *Method*.

3.3.3 *SCJ-mSafe* - Mission Sequencer

Mission sequencers are responsible for creating the missions of an SCJ program. This is achieved in the `getNextMission` method, which is defined as a specific component as it is called by the infrastructure.

As shown on line 18 of Figure 3.5, the mission sequencer definition has the basic class components plus a `getNextMission` method; the formalisation is shown below.

MissionSeq

$name : Name$
$fields : \operatorname{seq} Dec$
$init : \operatorname{seq} Com$
$constrs : \mathbb{P}\, Method$
$getNextMission : Com$
$methods : \mathbb{P}\, Method$

As in the *Safelet* schema, the *MissionSeq* schema includes components that capture the name, fields, field initialisation commands, and user-defined methods. In addition to these, the *getNextMission* component, which is of type *Com*, captures the `getNextMission` method.

3.3.4 *SCJ-mSafe* - Missions

Missions in *SCJ-mSafe* perform some initialisation tasks in the `initialize` method, executes the handlers it has defined, and then performs the `cleanUp` method after the handlers have finished executing. In addition to the basic class components, missions include the `initialize` and `cleanUp` methods that are specific to the paradigm as shown on line 26 of Figure 3.5. An extract of the mission from the ACCS example that shows the `initialize` method is shown in Figure 3.1. The corresponding formalisation of the mission is shown below.

Mission

$name : Name$
$fields : \operatorname{seq} Dec$
$init : \operatorname{seq} Com$
$constrs : \mathbb{P}\, Method$
$initialize : Com$
$handlers : \mathbb{P}\, Name$
$cleanUp : Com$
$methods : \mathbb{P}\, Method$

The *name*, *fields*, *init*, *constrs*, and *methods* components of the *Mission* schema are as defined in previous components. The additional components *initialize*, *handlers*, and *cleanUp*, define the `initialize` method, handlers of the mission, and `cleanUp` method, respectively.

3.3.5 *SCJ-mSafe* - Handlers

Handlers in *SCJ-mSafe* programs have the same basic components as above, but also include a specific `handleEvent` component, which is the command that is executed when the event handler has been registered by a mission and is triggered for execution (either periodically or by some external event). Handlers in *SCJ-mSafe* are not categorised as periodic or aperiodic, as the precise release point of a handler is not important in the checking technique.

The `handleEvent` method is defined as a specific component of the handler as shown on line 35 of Figure 3.5. A handler from the ACCS example is shown in Figure 3.2. It can not be called by other methods; it is only called during the execution phase of a mission when triggered. The corresponding formalisation of the handler is shown below.

Handler

$name : Name$
$fields : \operatorname{seq} Dec$
$init : \operatorname{seq} Com$
$constrs : \mathbb{P}\, Method$
$hAe : Com$
$methods : \mathbb{P}\, Method$

As above, the basic components of a class are included in the handler definition along with the `handleEvent` method (hAe), which is defined as a command (Com).

3.3.6 *SCJ-mSafe* - Classes

Classes in an *SCJ-mSafe* program are user-defined classes that are not part of the SCJ paradigm. Each one consists of the basic class components described above, as shown on line 1 of Figure 3.6. Classes also include information about the extended class where applicable. The corresponding definition in the formalisation is shown below.

Class

$$
\begin{array}{l}
name : Name \\
extends : Name \\
embeddedIn : Name \\
fields : \mathrm{seq}\, Dec \\
init : \mathrm{seq}\, Com \\
constrs : \mathbb{P}\, Method \\
methods : \mathbb{P}\, Method
\end{array}
$$

The *extends* component, which is of type *Name*, records the name of the extended class where applicable. The *embeddedIn* component records the name of the class in which the current class has been defined (again, if applicable), and is used when calculating the visible fields of *SCJ-mSafe* methods. This is required as embedded SCJ classes are no longer embedded in *SCJ-mSafe*; all classes are defined in the same way.

Interfaces are handled just like classes; this is because the typing issues of interfaces in Java is a concern of the compiler, and not important to the analysis here. Any classes in the input program that implement an interface must contain all of the necessary behaviour by definition, therefore if an interface type is used at any point, all possible classes that implement it are considered.

The definitions above capture the SCJ paradigm, where each element of the paradigm has its own definition; this is different to SCJ where each component would be treated as a regular class.

3.3.7 *SCJ-mSafe* - Methods

Methods in *SCJ-mSafe* are made up of a name, parameters, and a method body, as shown on line 8 of Figure 3.6. The SCJ method modifiers are no longer important; information about whether a method is public, private, or static, for example, is of no relevance. The definition of an *SCJ-mSafe* model in the formalisation is shown below.

$$
\begin{array}{|l}
\textit{Method} \\
\hline
name : Name \\
type : Type \\
returnType : Name \\
params : \mathrm{seq}\ Variable \\
body : Com \\
properties : MethodProperties \\
localVars : \mathbb{P}\ Expr \\
visibleFields : \mathbb{P}\ Expr \\
\hline
\end{array}
$$

Each method has a method name (*name*), the return type (*type*), which is of type *Type*, the return type name (*returnType*), a sequence of parameters (*params*), which is a sequence of *Variables*, and the actual body of the method (*body*), which is of type *Com*. In addition, *SCJ-mSafe* methods also have method properties (*properties*), which is of type *MethodProperties*, a set of local variables (*localVars*), which is of type *Expr*, and a set of visible fields (*visibleFields*), which is also of type *Expr*.

Method properties define the behaviour of a method independently of its execution; they are explained in greater detail in Section 4.3. The set of local variables is used when checking memory-safety of a method; it is necessary to check the variables defined inside a method for safety, but after which they are no longer required as they have gone out of scope. This set keeps a record of the expressions that need to be checked as part of the methods execution (including those defined in any subsequent method calls inside the body), but not in the scope of the method call. Finally, the set of visible fields records the fields of the method's containing class, and all fields of classes that the containing class extends in the class hierarchy. This is required to correctly identify which expressions are being referenced inside method bodies.

3.3.8 *SCJ-mSafe* - Commands

Commands in *SCJ-mSafe* include just a subset of those found in SCJ, as not all commands in SCJ affect memory safety. For example, the assert statement is not part of *SCJ-mSafe* as it has no impact on memory safety; similarly, the scope command is removed. Additional commands are included in *SCJ-mSafe*, however; SCJ expressions such as assignments, new instantiations, and method invocations are all represented as commands in *SCJ-mSafe*. They modify the value of program variables and are better characterised semantically as commands rather than expressions as in SCJ. The commands of *SCJ-mSafe* are shown on line 10 of Figure 3.6. The definition of an *SCJ-mSafe* command in the formalisation is shown below.

$$
\begin{aligned}
Com ::= {} & Skip \\
& \mid Asgn\langle\!\langle LExpr \times Expr\rangle\!\rangle \\
& \mid NewInstance\langle\!\langle newInstance\rangle\!\rangle \\
& \mid Seq\langle\!\langle Com \times Com\rangle\!\rangle \\
& \mid Decl\langle\!\langle Dec\rangle\!\rangle \\
& \mid DoWhile\langle\!\langle Com \times Expr\rangle\!\rangle \\
& \mid MethodCall\langle\!\langle methodCall\rangle\!\rangle \\
& \mid ExecuteInAreaOf\langle\!\langle MetaRefCon \times methodCall\rangle\!\rangle \\
& \mid ExecuteInOuterArea\langle\!\langle methodCall\rangle\!\rangle \\
& \mid EnterPrivateMemory\langle\!\langle methodCall\rangle\!\rangle \\
& \mid GetMemoryArea\langle\!\langle getMemoryArea\rangle\!\rangle \\
& \mid For\langle\!\langle Com \times Expr \times Com \times Com\rangle\!\rangle \\
& \mid If\langle\!\langle Expr \times Com \times Com\rangle\!\rangle \\
& \mid Switch\langle\!\langle Expr \times \mathrm{seq}\ Com\rangle\!\rangle \\
& \mid Try\langle\!\langle Com \times \mathrm{seq}\ Expr \times \mathrm{seq}\ Com \times Com\rangle\!\rangle \\
& \mid While\langle\!\langle Expr \times Com\rangle\!\rangle
\end{aligned}
$$

There are several differences between the commands in SCJ and *SCJ-mSafe*. As mentioned previously, assignments, new instantiations, and method calls are commands in *SCJ-mSafe*, whereas they are expression in SCJ. *SCJ-mSafe* also includes commands that identify SCJ-specific methods, such as *EnterPrivateMemory*, *ExecuteInAreaOf*, *ExecuteInOuterArea*, and *GetMemoryArea*. These are regular method calls in SCJ, however they are extracted as specific commands in *SCJ-mSafe* as they can directly affect memory safety. By recording the relevant information for each method call and defining them as individual commands, the memory safety of their uses can be checked statically without the need for knowledge of their implementation. Each *SCJ-mSafe* command is described individually below.

Skip The *Skip* command (line 10 of Figure 3.6) is new to *SCJ-mSafe* and defines the command with no behaviour; this is necessary if a command in SCJ is not needed in *SCJ-mSafe*. As demonstrated in Section 3.5, which defines the translation strategy, the process of translating an SCJ command must produce a corresponding *SCJ-mSafe* command; those commands not required are represented as *Skip*.

Assignment The assignment statement (line 11 of Figure 3.6) in *SCJ-mSafe* is the same as that in SCJ. For example, `a = b` is valid syntax for both SCJ and *SCJ-mSafe*.

The formalisation of the assignment command (*Asgn*) has a left expression of type *LExpr* and a right expression of type *Expr*. Left expressions are a subset of expressions that record references to objects or values. Expressions in *SCJ-mSafe* are explained after the commands.

NewInstance The *NewInstance* command (line 12 of Figure 3.6) is new to *SCJ-mSafe*, and replaces the `new` expression in SCJ. For example, the SCJ command `a = new A(b);` is defined as the *SCJ-mSafe* command `NewInstance(a, Current, A, (b));`. The arguments passed to the `NewInstance` command are the expression that is being assigned a reference to the new object, the memory area in which the new object is being created, which is the current memory area, the type of the object being created, and the arguments passed to the instantiation.

The `NewInstance` command is also used to represent the SCJ-specific method calls to the methods `newInstance` and `newArray`, which create new objects and new arrays in a specified memory area, respectively. New instantiations are defined by the *NewInstance* command, which has a *newInstance* parameter. The formalisation of the *newInstance* schema is shown below.

```
newInstance
  le : LExpr
  mrc : MetaRefCon
  type : VarType
  args : seq Expr
```

New instantiations are made up of the left expression that is being instantiated (*le*), which is of type *LExpr*, a meta-reference context (*mrc*), which describes the memory area in which the newly created object resides and is of type *MetaRefCon*, the type of the object (*type*), which is of type *VarType*, and the arguments passed to the appropriate constructor (*args*), which are a sequence of type *Expr*.

Meta-reference contexts are explained in greater detail later in Section 4.3, for now it is sufficient to think of them as a way of establishing which memory area the object being created resides. Most *NewInstance* commands will use the *Current* meta-reference context, which corresponds to the current memory area at the point the command is executed. The *VarType* construct that is used to describe the type of the object being instantiated is described with the *SCJ-mSafe* expressions after the commands.

Sequence The sequence command (line 13 of Figure 3.6) in *SCJ-mSafe* is used to represent a sequence of commands. For example, `a = 10; b = 5;` is a sequence that first associates `a` with the value 10 and then `b` with the value 5. The formalised *Seq* command has two commands of type *Com* in its definition, which correspond to the first and second commands in the sequence.

Declaration Declarations in *SCJ-mSafe* are the same as those in SCJ (line 14 of Figure 3.6), except that there can be no corresponding variable initialisation as part of the declaration. For example, the declaration `int a;` is the same in SCJ and *SCJ-mSafe*. The formalised *Decl* command has an associated declaration *Dec*, which is defined below.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Dec}} \\ \hline
\mathit{var} : \mathit{Variable} \\ \hline
\end{array}$$

The *Dec* schema has a single component, which is the variable *var* of type *Variable*. The definition of a *Variable* is presented with expressions later.

Do-while The do-while loop is the same in *SCJ-mSafe* as it is in SCJ (line 15 of Figure 3.6); for example, `do {a} while (b);` is valid in both SCJ and *SCJ-mSafe*. The corresponding *DoWhile* command in the formalisation has an associated command of type *Com* and expression of type *Expr*.

Method call Method calls in *SCJ-mSafe* (line 16 of Figure 3.6) are commands whereas they are expressions in SCJ. In SCJ, the returned result of a method call can be assigned to an expression, however, because method calls are commands in *SCJ-mSafe*, method calls to methods that have a return type other than `void` have an additional parameter, which is the result parameter. The returned result of the method call is assigned to the result parameter as part of the call. For example, the method call `a = b.get();` in SCJ is represented as `b.get(a);` in *SCJ-mSafe*.

The method call command *MethodCall* in the formalisation has an associated *methodCall* parameter; the *methodCall* schema is defined below.

$$
\begin{array}{|l}
\textit{methodCall} \\
\hline
le : LExpr \\
name : Name \\
args : \mathrm{seq}\, Expr \\
methods : \mathbb{P}\, MethodSig \\
\hline
\end{array}
$$

Method calls are made up of a left expression (*le*) of type *LExpr*, which is the expression that identifies the target object of the call, the name of the actual method (*name*) of type *Name*, the arguments (*args*), which is a sequence of expressions of type *Expr*, and a set of method signatures (*methods*) of type *MethodSig* that identifies the possible methods being called.

Method signatures define a syntactic summary of the possible methods associated with a method call. As will be explained later in the translation, these are required when extracting embedded method calls from complex statements that require the introduction of new variables, they are also required to handle dynamic binding. Without method signatures, it is not possible to determine the type of the new variable that is being introduced. For example, the SCJ expression `method1(method2(), a, b);` passes the result of `method2` as an argument to `method1`. As there are no embedded statements in *SCJ-mSafe*, the resulting sequence of *SCJ-mSafe* commands is a new declaration to store the result of `method2`, followed by the call to `method2` with the newly defined variables as the result parameter, and then finally the call to `method1` with the newly defined variable, `a`, and `b` as its arguments: `retTypeM1 var; method1(var); method2(var, a, b);`

ExecuteInAreaOf The `executeInAreaOf` method in SCJ (line 17 of Figure 3.6) takes two arguments: a runnable object and an expression, whose memory area is used as the execution area for the runnable object.

Figure 3.7 shows an example of how the `executeInAreaOf` method is used in SCJ. The `handleAsyncEvent` method of the handler creates a new instance of the runnable class `MyRunnable`, and assigns it to the `myRun` variable. The runnable object is later executed in the memory area of the object referenced by the variable `field` through the `executeInAreaOf` method call. In this simple example, a reference to the local variable `data` is passed to the runnable object, and is stored as a local reference in the field `runField`; this is perfectly legal. When the runnable object is executed through the `executeInAreaOf` method call, the field of the handler (`field`) is assigned to point to the local field of the runnable class `runField`. This introduces a memory-safety violation as

```java
public class MyHandler extends PeriodicEventHandler {

    A field = new A();

    ...

    public void handleAsyncEvent() {
        B data = new B();
        MyRunnable myRun = new MyRunnable(data);
        ManagedMemory.executeInAreaOf(field, myRun);
    }

    class MyRunnable implements Runnable {
        A runField;

        public MyRunnable(A arg) {
            runField = arg;
        }
        public void run() {
            field = runField;
        }
    }
}
```

Figure 3.7: executeInAreaOf example in SCJ

the handler field now references an object that resides in the per-release memory area of the handler, whereas it should only reference objects stored in the mission memory area or higher.

As demonstrated, it is possible for the `executeInAreaOf` method to introduce memory-safety violations; for this reason it is defined as an individual command in *SCJ-mSafe*. Figure 3.8 shows the same example in *SCJ-mSafe*. The translation shows the specific `ExecuteInAreaOf` command, which states that the runnable object referenced by `myRun` will be executed in the memory area associated with the handler field (`Erc field`). The construct `Erc field` is a meta-reference context, which defines the memory area of an object independently of the execution of the program, and is described in more detail in Section 4.3.

The formalisation of the *ExecuteInAreaOf* command has a meta-reference context (*MetaRefCon*) and a method call (*methodCall*) as its parameters. The meta-reference context is used to describe the memory area in which the `run` method described by the *methodCall* component is executed. Meta-reference contexts are explained in more detail in Section 4.3. The *ExecuteInAreaOf* command uses a *methodCall* in *SCJ-mSafe* as opposed to a reference to a runnable object like in SCJ as it is the `run` method of the runnable

```
handler MyHandler {
  fields {
    A field;
  }
  init {
    NewInstance(field, Current, A, ());
  }
  ...
  handleEvent {
    B data;
    NewInstance(data, Current, B, ());
    MyRunnable myRun;
    NewInstance(myRun, Current, MyRunnable, (data));
    ExecuteInAreaOf(Erc field, myRun);
  }
}

class MyRunnable {
  fields {
    Object runField;
  }
  init {
  }
  constr (arg) {
    runField = arg;
  }
  method run() {
    field = runField;
  }
}
```

Figure 3.8: executeInAreaOf example in *SCJ-mSafe*

object that is called as a result of its execution; the possibility of dynamic binding is still handled. Therefore, as will be described during the translation strategy in Section 3.5, the corresponding method call to the `run` method of the target object is defined.

ExecuteInOuterArea The `executeInOuterArea` method call in SCJ (line 18 of Figure 3.6) is very similar to the `executeInAreaOf` method presented above. The difference between the two commands is that the `executeInAreaOf` command specifies the exact area in which a runnable object will execute, whereas `executeInOuterArea` always executes the runnable object in the immediate outer memory area, which is established based on the current area and the memory hierarchy defined in the previous chapter.

Calls to the `executeInOuterArea` method in SCJ are defined as *ExecuteInOuterArea* commands in *SCJ-mSafe*, which have a *methodCall* component as their parameter.

```
public class MyHandler extends PeriodicEventHandler {

    A handlerField = new A();

    public void handleAsyncEvent() {
        B data;
        MemoryArea memArea =
           MemoryArea.getMemoryArea(handlerField);
        data = memArea.newInstance(B.class);
    }
}
```

Figure 3.9: getMemoryArea example in SCJ

EnterPrivateMemory The `enterPrivateMemory` method (line 19 of Figure 3.6) is similar to the `executeInOuterArea` method described above; however, instead of executing a runnable object in the memory area immediately outside the current scope, the `enterPrivateMemory` method call creates a new temporary private memory area specifically for the runnable object to execute. This can only be done during the initialisation phase of the mission, and during the `handleEvent` methods of handlers.

If a reference to an object is passed as an argument to the runnable class, and the corresponding `run` method manipulates fields of that reference, it is possible to introduce downward references, which are illegal.

Calls to the `enterPrivateMemory` method in SCJ are defined as *EnterPrivateMemory* commands in *SCJ-mSafe*, which have a *methodCall* component as their parameter.

GetMemoryArea The `getMemoryArea` method in SCJ (line 20 of Figure 3.6) is used to acquire a reference to the memory area in which a particular object resides. Once a reference has been established, it is possible to call the `newInstance` or `newArray` methods, which create new objects and arrays respectively, of a specific type, in the specific memory area. This illustrates that it is possible to create objects in memory areas other than the current one during execution. Figure 3.9 shows a small example of how the `getMemoryArea` method can be used to create a new instance of an object in a separate memory area.

The object referenced by the field of the handler (`handlerField`) is located in mission memory as it was instantiated when the handler object was created. The local variable `data` in the `handleAsyncEvent` method would reference an object in the handler's per-release memory area if instantiated normally. However, because the `newInstance` method is used to create the new object, the object referenced by `data` is instantiated in the memory area referenced by `memArea`, which is the same memory area as `handlerField`,

```
handler MyHandler {
  fields {
    A a;
  }
  init {
    NewInstance(a, Current , A, ());
  }
  handleEvent {
    B data;
    MemoryArea memArea;
    GetMemoryArea(memArea , a);
    NewInstance(data , Erc memArea , B, ());
  }
}
```

Figure 3.10: getMemoryArea example in *SCJ-mSafe*

because the local variable `memArea` is assigned a reference to the mission memory area through the `getMemoryArea` method.

As the `getMemoryArea` method can have a direct impact on where objects are instantiated, it is defined as a specific command in *SCJ-mSafe*; Figure 3.10 shows the same example in *SCJ-mSafe*. The method call `MemoryArea.getMemoryArea` in SCJ is defined as the *SCJ-mSafe* command `GetMemoryArea`; the memory area in which the object referenced by variable `a` resides is stored into the local variable `memArea`.

The SCJ method call to `memArea.newInstance` is defined as a `NewInstance` command in *SCJ-mSafe*. The meta reference context that determines which memory area the object is instantiated is the memory area of `memArea`; this is represented as `Erc memArea`.

The formalised *GetMemoryArea* command has a *getMemoryArea* parameter, which is shown below.

getMemoryArea

$ref : LExpr$

$e : LExpr$

The *ref* component of *getMemoryArea* identifies the expression that is being queried to determine the memory area, and the e component represents the expression that is assigned the result.

For loop The for loop is the same in *SCJ-mSafe* as it is in SCJ (line 21 of Figure 3.6); for example, `for(a; b; c) {d}` is valid in both SCJ and *SCJ-mSafe*. The corresponding *For* command in the formalisation has three commands and an expression as its parameters.

The commands, which are of type *Com*, represent the initialisation command, iteration command, and body of the loop, respectively. The expression, which is of type *Expr*, represents the loop condition that must be true in order for the body to be executed.

If Conditional statements are also the same in *SCJ-mSafe* as they are in SCJ (line 22 of Figure 3.6); for example, `if(a) {b} else {c}` is valid in both SCJ and *SCJ-mSafe*. Although the syntax and meaning of the statement are the same, conditional statements in *SCJ-mSafe* always have an `else` branch, even if the behaviour of the `else` branch is simply *Skip*; this is to keep the structure of commands uniform.

The corresponding *If* command in the formalisation has an expression and two commands as its parameters. The expression, which is of type *Expr*, is the condition that must be true for the true branch to be executed; the false (or `else`) branch is executed otherwise. Although the expression is not required for the analysis, it is maintained in *SCJ-mSafe* to ease readability; this is also the case for subsequent commands presented. The two commands, which are of type *Com*, represent the true and false branches respectively.

Switch Switch statements are subtly different in *SCJ-mSafe* to those in SCJ (line 23 of Figure 3.6), the difference is that the expressions that are used to identify the cases in SCJ are removed in *SCJ-mSafe*. This is because it is not possible to determine statically which case will be executed, therefore, the possible cases are simply defined based on their commands alone.

For example, the SCJ command `switch(e) { case(1) {a}; case(2) {b} ...}` is represented in *SCJ-mSafe* as `switch(e) { {a}, {b} }`.

The corresponding *Switch* command in the formalisation has an expression and a sequence of commands as its parameters. The expression, which is of type *Expr*, is the expression that is being compared to determine which case is to be executed. The sequence of commands, which is of type *Com*, represents the commands of the cases that may be executed.

Try Try statements are the same in *SCJ-mSafe* as they are in SCJ (line 24 of Figure 3.6); for example, `try{a} catch(b){c} finally{d}` is the same in *SCJ-mSafe* as it is in SCJ. The corresponding *Try* command in the formalisation has four parameters; the first is the command, which is of type *Com*, that represents the initial command whose execution is attempted. The next two parameters, which are a sequence of expressions and a sequence of commands, represent the possible catch expressions and statements; these are of type *Expr* and *Com* respectively. The fourth parameter is the finally command, which is of type *Com*, that is executed when the try block exits.

While Finally, the *While* command is also the same in *SCJ-mSafe* and SCJ (line 25 of Figure 3.6); for example, `while(a) {b}` is the same in *SCJ-mSafe* as it is in SCJ. The corresponding *While* command in the formalisation has an expression, which is of type *Expr*, and a command, which is of type *Com*, as its parameters. These represent the condition that must be true in order for the body of the loop to execute, and the body of the loop, respectively.

3.3.9 *SCJ-mSafe* - Expressions

Some SCJ expressions are not required in *SCJ-mSafe* as they are not able to affect memory safety based on the static analysis technique used here; for example, an expression such as `++count` may be crucial to behaviour and subsequently impact the memory configuration of a program, but has no relevance to memory safety here as all execution paths are considered regardless of the result of an expression. The syntax of expressions in *SCJ-mSafe* is shown on lines 27 to 35 of Figure 3.6. The definition of all possible expressions include values, identifiers, field accesses, other expressions, the special value null, and the keyword this.

$$Expr ::= Val \mid ID\langle\langle Identifier\rangle\rangle \mid FA\langle\langle FieldAccess\rangle\rangle \mid OtherExpr \mid Null \mid This$$

The *OtherExpr* expression is used to identify expressions in SCJ that are not relevant to memory safety, and therefore not included in *SCJ-mSafe*. For example, the expression `x > y` may be used in a conditional statement to determine which branch is executed, however, this is not relevant in *SCJ-mSafe*, and is defined as an *OtherExpr*.

The important expressions in *SCJ-mSafe* are left expressions, which are expressions that can reference objects; identifiers and field accesses are left expressions as shown on line 31 of Figure 3.6. Left expressions in *SCJ-mSafe* are a subset of the possible expressions, and are defined as the union of valid identifiers and valid field accesses.

$$LExpr == \operatorname{ran} ID \cup \operatorname{ran} FA$$

Identifiers and field accesses denote objects manipulated in a program whose allocations need to be checked. An identifier is a variable or an array access, as shown on line 33 of Figure 3.6.

$$Identifier ::= var\langle\langle Variable\rangle\rangle \mid arrayElement\langle\langle ArrayElement\rangle\rangle$$

Field accesses are defined as a sequence of identifiers with at least two elements.

$$FieldAccess == seqtwo[Identifier]$$

The *seqtwo* definition ensures that all field accesses are sequences of identifiers whose length is at least two, as there must be at least two elements in a field access; otherwise it is just an identifier.

Variables are made up of a name, and a *VarType*.

```
Variable
  name : Name
  varType : VarType
```

The name is the variable's identifier, and the *VarType* records information about the type of the variable.

```
VarType
  type : Name
  isArray : Boolean
  isPrimitive : Boolean
  isReference : Boolean
  resultVar : Boolean
  ----------
  isPrimitive ≠ isReference
```

The type information recorded about each variable includes the type name, and the category of the type, which is sufficient to distinguish how the variable is handled later in the checking procedure; for example, the variable may or may not be an array, and may be a primitive or reference type. The definition includes an invariant that states a variable cannot be of both reference and primitives types simultaneously. Variables may also be result variables.

Identifiers can also be array accesses, which are modelled as *ArrayElements*; these are made up of a name and a type.

```
ArrayElement
  name : Name
  type : Name
```

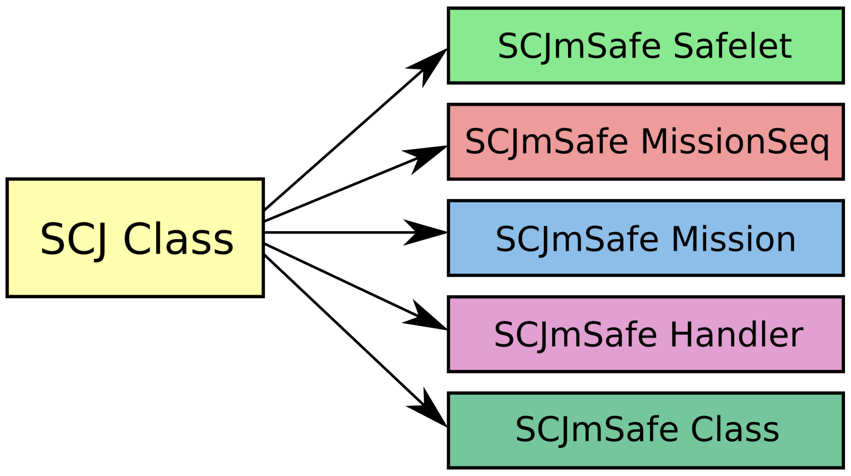


Figure 3.11: SCJ class translation to *SCJ-mSafe* paradigm components.

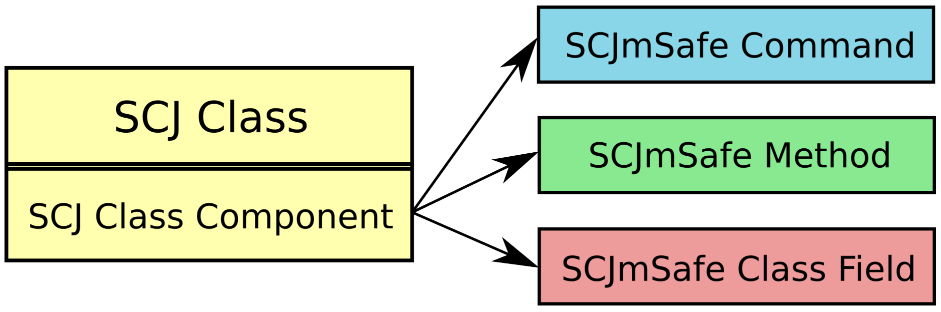


Figure 3.12: SCJ class component translation to *SCJ-mSafe* methods and class fields.

This model of *SCJ-mSafe* captures all of the information required to perform the analysis and checking procedures to determine whether a program is memory safe or not. The next sections describe the translation from SCJ to *SCJ-mSafe*.

3.4 Translating SCJ to *SCJ-mSafe*

Previous sections have described SCJ and *SCJ-mSafe*, this section describes the overall translation from SCJ to *SCJ-mSafe*.

The translation from SCJ programs to *SCJ-mSafe* is not trivial, and includes analysis of the input program to create an *SCJ-mSafe* program with the uniform structure required for analysis. The translation is defined by a series of mappings from SCJ components to corresponding *SCJ-mSafe* components.

The input SCJ program must be well formed and well typed; more specifically, no errors must be raised during compilation. The program must correspond to the SCJ specification [46], and be defined in separate classes that identify the Level 1 SCJ programming paradigm. Whilst it is possible to extend the translation to refactor input programs that

```
program {
  static {
    ...
  }
  sInit {
    ...
  }
  safelet {
    ...
  }
  missionSeq {
    ...
  }
  mission ACCMission {
    ...
  }
  handler SpeedMonitor {
    ...
  }
  handler Engine {
    ...
  }

  ...

  class Controller {
    ...
  }
}
```

Figure 3.13: ACCS sketch in *SCJ-mSafe*

are not defined in separate classes, this restriction is not unfair given the guidelines in the specification.

The input SCJ program consists of a series of SCJ classes. Classes that make up the SCJ programming paradigm are translated into the corresponding *SCJ-mSafe* component described previously. Classes that are user-defined, that is, they are not part of the SCJ programming paradigm, are translated into regular *SCJ-mSafe* classes. Figure 3.11 shows a summary of this translation, and indicates that classes in *SCJ-mSafe* are separated into individual classes based on the paradigm of SCJ.

The components of SCJ classes are either methods or class fields; these are translated to *SCJ-mSafe* methods, class fields, or commands. The SCJ methods that define the programming paradigm, as mentioned above, are translated to *SCJ-mSafe* commands; for example, the `initialize` method of missions is defined as a command (Com) in *SCJ-mSafe*. Figure 3.12 indicates that SCJ class components are translated into three possible categories.

Figure 3.13 shows a sketch of the ACCS in *SCJ-mSafe*. Although very simple, this gives an idea of the layout and structure of an overall *SCJ-mSafe* program; the ACCS mission (`ACCMission`) and speed monitor handler (`SpeedMonitor`) defined on lines 14 and 17 respectively were presented in greater detail in Figures 3.1 and 3.2.

As shown from the program outline, components of the SCJ paradigm are defined in an abstract way; for example, there is no need for package definitions or library classes to be imported. Similarly, annotations and modifiers are not maintained; this is because the input SCJ program is well-formed and type-correct, therefore the validity of the program does not need to be re-checked.

3.5 A translation strategy

The translation strategy is defined by a series of compositional functions that map SCJ components to corresponding *SCJ-mSafe* components. The formalisation of the translation functions is described here; the full version can be found in Appendix D. The top-level *Translate* function, which translates the overall program, takes an SCJ program and returns an *SCJ-mSafe* program; it is shown in Figure 3.14.

The SCJ programs in the domain of *Translate* are a subset of SCJ programs that are valid and well typed (*WellTypedPrograms*), as defined on line 3.

Line 4 of the function defines the resulting *SCJ-mSafe* program (*scjmsafe*) and a translation environment (*transEnv*). Lines 6-36 define the components of the resulting *SCJ-mSafe* program.

The translation environment records the set of *SCJ-mSafe* variables defined from SCJ variables throughout the translation, and also the set of method signatures for the program.

TranslationEnv

$variables : \mathbb{P}\ Variable$
$methods : \mathbb{P}\ MethodSig$

As explained previously, method signatures record a summary about each method in the SCJ program and are used to handle overloading and to establish which method is being called at each method call. Method signatures are required at translation time to facilitate the introduction of new variables, where necessary, when extracting embedded

```
$Translate : SCJProgram \pfun SCJmSafeProgram$
─────────
$\forall program : SCJProgram$
    $| \; program \in WellTypedPrograms$
    $\bullet \exists scjmsafe : SCJmSafeProgram; \; transEnv : TranslationEnv$
        $| \; transEnv.methods = AnalyseMethodSigs \; program$
        $\bullet (\exists scjSafelet : SCJClass$
            $| \; scjSafelet \in program.classes$
            $\land Extends(scjSafelet, safelet, program) = True$
            $\bullet scjmsafe.safelet = TranslateSafelet(scjSafelet,$
                    $scjmsafe, transEnv))$
        $\land (\exists scjMissionSeq : SCJClass$
            $| \; scjMissionSeq \in program.classes$
            $\land Extends(scjMissionSeq, missionSequencer, program) = True$
            $\bullet scjmsafe.missionSeq =$
                $TranslateMissionSeq(scjMissionSeq, scjmsafe, transEnv))$
        $\land (\forall scjMission : SCJClass$
            $| \; scjMission \in program.classes$
            $\land Extends(scjMission, mission, program) = True$
            $\bullet TranslateMission(scjMission, program, scjmsafe, transEnv)$
                $\in scjmsafe.missions)$
        $\land (\forall scjHandler : SCJClass$
            $| \; scjHandler \in program.classes$
            $(\land Extends(scjHandler, APeriodicHandler, program) = True$
                $\lor Extends(scjHandler, PeriodicHandler, program) = True)$
            $\bullet TranslateHandler(scjHandler, scjmsafe, transEnv)$
                $\in scjmsafe.handlers)$
        $\land (\forall scjClass : SCJClass$
            $| \; scjClass \in program.classes$
            $\land abstract \notin scjClass.modifiers.flags$
            $\land Extends(scjClass, safelet, program) = False$
            $\land Extends(scjClass, missionSequencer, program) = False$
            $\land Extends(scjClass, mission, program) = False$
            $\land Extends(scjClass, APeriodicHandler, program) = False$
            $\land Extends(scjClass, PeriodicHandler, program) = False$
            $\bullet TranslateClass(scjClass, scjmsafe, transEnv)$
                $\in scjmsafe.classes)$
        $\land Translate \; program = scjmsafe$
```

Figure 3.14: *Translate* function that takes SCJ programs and returns *SCJ-mSafe* programs.

method calls from complex statements.

Each method signature consists of the method name, the class in which it is defined, the name of the class that the method's class extends if applicable, a set of class names that extend the method's class (its descendants), the method return type and return type name, and a sequence of type names for the parameters. A simple syntactic function can be defined to specify how method signatures can be identified from the text of an SCJ program, as needed to construct the translation environment.

$$\begin{array}{l}
\textit{MethodSig} \\
\hline
\textit{name} : \textit{Name} \\
\textit{class} : \textit{Name} \\
\textit{classExtends} : \textit{Name} \\
\textit{descendants} : \mathbb{P}\,\textit{Name} \\
\textit{returnType} : \textit{Type} \\
\textit{returnTypeName} : \textit{Name} \\
\textit{paramTypes} : \operatorname{seq} \textit{Name} \\
\hline
\end{array}$$

Most components of method signatures are of type *Name* because method signatures give a syntactic description of the methods of a program before the translation.

All classes in an SCJ program are recorded inside the program component *program.classes*; there is no distinction at the SCJ level between the safelet, mission sequencer, missions, or handlers. The overall *Translate* function includes a case analysis for each SCJ component; if a class in *program.classes* matches the criteria for a specific component, the respective function to translate each component is used. Lines 6-10 define the criteria for the safelet component, lines 11-15 define mission sequencers, lines 16-20 define missions, lines 21-26 define handlers, and lines 27-36 define user-defined classes.

The *TranslateHandler* function is used on line 25 when the result of the *Extends* function, which checks whether a class extends another with a particular name in the class hierarchy, is true when compared against the names *APeriodicHandler* and *PeriodicHandler*. The function that translates SCJ classes identified as handlers into *SCJ-mSafe* is shown in Figure 3.15.

On line 5, the function specification states that there must exist a handler (*handler*) of type *Handler*, which is the *SCJ-mSafe* handler identified as the result of the function on line 31. Lines 6-30 identify the properties of the handler. The fields (*handler.fields*) are defined on lines 6-8, for example. The corresponding field initialisation commands on lines 9-11, the constructors on lines 12-17, the `handleEvent` method on lines 18-22, any

```
1  $TranslateHandler : SCJClass \times SCJmSafeProgram$
2        $\times\ TranslationEnv \rightarrow Handler$
   ----------------------------------------------------
3  $\forall scjClass : SCJClass;\ program : SCJmSafeProgram;$
4        $transEnv : TranslationEnv$
5     $\bullet \exists handler : Handler$
6        $\bullet handler.fields =$
7           $TranslateComponentsFieldsDecs(scjClass.members,$
8                 $program, transEnv)$
9        $\land handler.init =$
10          $TranslateComponentsFieldsInits(scjClass.members,$
11                $program, transEnv)$
12       $\land handler.constrs =$
13          $\bigcup\{classComponent : \mathrm{ran}\ scjClass.members$
14             $\bullet \{method : SCJMethod$
15                $\mid classComponent = ClassMethod\ method$
16                $\land method.name = handler.name$
17                $\bullet (TranslateConstr(method, program, transEnv))\}\}$
18       $\land (\exists classComponent : \mathrm{ran}\ scjClass.members;\ method : SCJMethod$
19          $\mid classComponent = ClassMethod\ method$
20          $\land method.name = handleEvent$
21          $\bullet handler.hAe = CreateSingleCommand($
22             $TranslateCommandSeq(method.body, program, transEnv)))$
23       $\land handler.methods =$
24          $\bigcup\{classComponent : \mathrm{ran}\ scjClass.members$
25             $\bullet \{method : SCJMethod$
26                $\mid classComponent = ClassMethod\ method$
27                $\land method.name \neq handler.name$
28                $\land method.name \neq handleEvent$
29                $\bullet (TranslateMethod(method, program, transEnv))\}\}$
30       $\land handler.name = scjClass.name$
31       $\land TranslateHandler(scjClass, program, transEnv) = handler$
```

Figure 3.15: *TranslateHandler* function that takes an SCJ class identified as a handler, and returns an *SCJ-mSafe* handler.

user-defined methods on lines 23-29, and the handler name on line 30.

The *TranslateHandler* function calls subsequent functions to translate individual aspects of the handler. For example, the function *TranslateComponentsFieldsDecs* used on line 7 takes a sequence of SCJ class members ($scjClass.members$), the resulting *SCJ-mSafe* program ($program$), and the translation environment ($transEnv$) as its parameters, and characterises a sequence of declarations that identify the fields of the handler; these are recorded in the *handler.fields* component.

The corresponding initialisation commands for the fields, if any, are identified and identified in the *handler.init* component with the *TranslateComponentsFieldsInits* function on line 10, which takes the same arguments as above, and defines a sequence of *SCJ-mSafe* commands.

Constructors and methods of the handler are also SCJ class components recorded in *scjClass.members*; constructors are identified as methods whose name corresponds to the name of the handler ($method.name = handler.name$ on line 16), and are recorded in *handler.constrs*. The constructors are translated with the *TranslateConstr* function on line 17, which takes the SCJ method identified as a constructor (*method*), the resulting *SCJ-mSafe* program (*program*), and the translation environment (*transEnv*), and returns an *SCJ-mSafe* method.

The special method `handleEvent` is recorded as a specific component of the handler (*handler.hAe*), and is identified through the condition $method.name = handleEvent$ on line 20. The SCJ method body (*method.body*) is translated using the *TranslateCommandSeq* function, which takes a sequence of SCJ commands and returns a sequence of *SCJ-mSafe* commands. This sequence is then combined into a single command that makes use of the dedicated sequence command *Seq* with the *CreateSingleCommand* function; the resulting command that defines the behaviour of the `handleEvent` method of the handler is recorded in *handler.hAe*.

Additional methods that are not constructors or the special `handleEvent` method are recorded in the handler component *handler.methods*; these are translated with the *TranslateMethod* function, which is similar to the *TranslateConstr* function, except that it introduces the additional result variable for methods that return a reference or value.

3.5.1 Translating expressions

Expressions in SCJ that identify values or references are translated into expressions in *SCJ-mSafe*; the remaining expressions that impact memory safety are translated to commands. Accordingly, two translation functions for expressions are defined.

TranslateExpression The first defines the translation of expressions into commands (*TranslateExpression*). This function takes an SCJ expression and returns an *SCJ-mSafe* command, and is shown in Figure 3.16.

```
1  $TranslateExpression : SCJExpression \times TranslationEnv \nrightarrow Com$
   ----------
2  $\forall scjExpr : SCJExpression; transEnv : TranslationEnv$
3      $\mid (scjExpr, transEnv) \in \mathrm{dom}\ TranslateExpression$
4      $\land scjExpr \in WellTypedExprs$
5        $\bullet \ldots$
6        $\lor (\exists e1, e2 : SCJExpression; lexpr, rexpr : Expr$
7          $\mid scjExpr = assignment(e1, e2)$
8          $\land lexpr = ExtractExpression\ e1$
9          $\land rexpr = ExtractExpression\ e2$
10           $\bullet \ldots$
11           $\lor (\exists e3, e4 : SCJExpression; args : \mathrm{seq}\ SCJExpression;$
12             $name : Name; type : TypeElement$
13               $\mid e2 \neq newArray(type, args)$
14               $\land e2 \neq newClass(name, args)$
15               $\land e2 \neq methodInvocation(e3, args)$
16               $\land e2 \neq arrayAccess(e3, e4)$
17               $\bullet TranslateExpression(scjExpr, transEnv) =$
18                 $SimplifyCommandPair($
19                     $(TranslateExpression(e2, transEnv)),$
20                     $(Asgn(lexpr, rexpr)))))$
```

Figure 3.16: *TranslateExpression* function that illustrates the translation of a simple assignment.

The SCJ expressions in the domain of *TranslateExpression* are a subset of SCJ expressions that are valid and well typed (*WellTypedExprs*) as shown on line 4; for all SCJ expressions in its domain, the resulting *SCJ-mSafe* command is defined based on a case analysis of the type of expression. The simplest case for assignments, where the right-hand side is not a new instantiation, method call, or array access, is shown above.

The SCJ assignment is identified on line 7; the *ExtractExpression* function, which ignores any embedded commands inside expressions and simply defines the result of the expression, is applied to the left and right-hand sides of the assignment on lines 8 and 9 respectively. The resulting expressions are recorded in *lexpr* and *rexpr*. The *ExtractExpression* function is described later.

Lines 13-16 are used to identify the type of expression being translated; in this case, it is a simple assignment where the right-hand side ($e2$) is not a new array, new class, method invocation, or array access expression. Side effects embedded in the right-hand side of the assignment statement are handled with a recursive call to the *TranslateExpression* function on line 19. The resulting command from this call is passed as the first parameter to the *SimplifyCommandPair* function on line 18, which takes two separate commands and defines a single command that uses the sequence command *Seq*. The second command passed to *SimplifyCommandPair* is the new *SCJ-mSafe* assignment command on line 20.

In this way, assignments such as `a = (b = c)`, which contain side effects, are translated as a sequence (*Seq*) of commands. The result of applying *TranslateExpression* to `a = (b = c)` is $Seq(Asgn(b, c), Asgn(a, b))$. This is done by translating any embedded side effects into separate commands that come first in a sequence, followed by the overall expression; `b = c` is an embedded side effect of `a = (b = c)`.

If an expression has no embedded side effects, the result of *TranslateExpression* on line 19 is the command *Skip*. For example, the SCJ assignment `a = b` has no side effects and is translated into the sequence *Skip* followed by $Asgn(a, b)$. The introduction of the command *Skip* is later removed by the *SimplifyCommandPair* function on line 18, which simplifies the translation by removing unnecessary *Skip* commands.

The translation of assignments becomes significantly more complex if the right-hand side of the assignment is a new instantiation, method call, or array access. As an example, consider the instantiation of a new object: in SCJ this is treated as an assignment, where the right-hand side includes the keyword `new`; this is translated into a *NewInstance* command in *SCJ-mSafe* shown in Figure 3.17. If a new instantiation is embedded inside another expression, a sequence of commands is defined in *SCJ-mSafe* that records the new instantiation and overall expression separately.

This extract from *TranslateExpression* specifies how a new instance command *nI* is constructed from the SCJ assignment (*scjExpr* on line 3) and new class instantiation ($e2$ on line 8). The type of the new object being instantiated is equal to the *name* of the new class, as shown on line 9. The left expression (*lexpr*) being assigned the reference to the new object is recorded in the new instance component *nI.le*, as shown on line 10. The instantiation is taking place in the current memory area, therefore the meta-reference context (*nI.mrc*) of the new instance, as described previously, is *Current*, as shown on line 11. The arguments passed to the instantiation of the new class are recorded in the new instance component *nI.args*, as shown on line 12; this is a sequence of expressions passed to the *ExtractExpression* function.

The overall result of the *TranslateExpression* function on line 13 in this scenario is the result of the *MergeSideEffectsParamsCom* function, which produces a *Seq* command

```
...
∨ ($\exists e1, e2 : SCJExpression;\ lexpr, rexpr : Expr$
   $\mid scjExpr = assignment(e1, e2)$
   $\land lexpr = ExtractExpression\ e1$
   $\land rexpr = ExtractExpression\ e2$
      $\bullet$ ...
      $\lor (\exists args : \mathrm{seq}\ SCJExpression;\ nI : newInstance;\ name : Name$
         $\mid e2 = newClass(name, args)$
         $\land nI.type.type = name$
         $\land nI.le = lexpr$
         $\land nI.mrc = Current$
         $\land nI.args = \{n : 1 .. \# args \bullet n \mapsto ExtractExpression(args\ n)\}$
         $\bullet\ TranslateExpression(scjExpr, transEnv) =$
            $MergeSideEffectsParamsCom((NewInstance\ nI),$
               $\{n : 1 .. \# args$
                  $\bullet (n \mapsto TranslateExpression((args\ n), transEnv))\}, Skip))$
```

Figure 3.17: *TranslateExpression* function that illustrates a new instantiation as part of an assignment.

based on the side effects of the arguments passed to the new instantiation, and the new instance command itself. Any side effects in the arguments to the new instantiation are extracted through the *TranslateExpression* function, and are placed in a sequence before the *SCJ-mSafe* new instance command $NewInstance\ nI$.

For example, the simple assignment `a = new A(x = y);` is translated to $Seq(Asgn(x, y), NewInstance(a, Current, A, (x)))$. The side effect in the argument passed to the new instantiation (`x = y`) is translated as a separate assignment command and placed in a sequence before the *NewInstance* command. The *NewInstance* command then states that the left expression a is assigned a new object of type A in the *Current* memory area, with arguments x.

If the right-hand side of the assignment is a method call, the translation first analyses the method call, and then adds the left expression of the assignment as the result parameter of the method call. If a method call is embedded inside a more complex expression, it is extracted; if necessary, new variables are introduced to record the values of method calls that have been extracted from more complex expressions. These newly defined variables are then used in place of the existing method call in the containing expression.

```
1  ...
2  $\lor (\exists e1 : SCJExpression; args : \operatorname{seq} SCJExpression; name, mname : Name;$
3     $c, sideEffect : Com; lexpr : Expr; paramComs : \operatorname{seq} Com;$
4          $paramExprs : \operatorname{seq} Expr$
5        $\mid scjExpr = methodInvocation(e1, args)$
6        $\land c = TranslateExpression(e1, transEnv)$
7        $\land paramComs = \{n : 1 .. \# args$
8             $\bullet n \mapsto TranslateExpression((args\ n), transEnv)\}$
9        $\land paramExprs = \{n : 1 .. \# args$
10            $\bullet n \mapsto ExtractExpression(args\ n)\}$
11       $\land mname = GetMethodName\ e1$
12       $\bullet ((\exists c1, c2 : Com; mc : methodCall$
13         $\mid c = Seq(c1, c2) \land c2 = MethodCall\ mc$
14         $\lor c = MethodCall\ mc$
15         $\bullet lexpr = (GetMethodCallReturnDec(mc, c)).1$
16            $\land sideEffect = (GetMethodCallReturnDec(mc, c)).2)$
17       $\lor (\exists c1, c2 : Com; mc : methodCall \mid$
18         $c \neq MethodCall\ mc$
19         $\land c \neq Seq(c1, c2)$
20         $\bullet lexpr = ExtractExpression\ e1$
21            $\land sideEffect = c))$
```

Figure 3.18: *TranslateExpression* function that illustrates the initial translation of method invocations.

Method calls are the most complex part of the *TranslateExpression* function; the first part of its formalisation is shown in Figure 3.18. The left-hand side of the method call ($e1$) needs to be analysed, as this may include side effects such as additional method calls; this is shown on line 6. The command c records the result of applying *TranslateExpression* to the left-hand side of the method call $e1$. The *paramComs* component records a sequence of commands that represent the side effects in the parameters of the method call; this is shown on lines 7 and 8. The *paramExprs* component records a sequence of expressions that ignore the extracted side effects, as shown on lines 9 and 10.

The resulting left expression (*lexpr*) and side effect (*sideEffect*) components are based on the result of c. If c is another method call, or a sequence whose second element is a method call (because the first element is the side effect of the embedded method call), then *lexpr* and *sideEffect* are specified using the *GetMethodCallReturnDec* function on lines 15 and 16. This function analyses the method call and returns a command and an

```
1  ...
2  $\lor\ mname = enterPrivateMemoryID$
3      $\land\ (\exists\ mc : methodCall$
4          $\mid mc.methods = FindMethods((paramExprs\ 1), run, \langle\rangle, transEnv)$
5          $\land\ mc.le = paramExprs\ 1$
6          $\land\ mc.args = \langle\rangle$
7          $\land\ mc.name = run$
8          $\bullet\ TranslateExpression(scjExpr, transEnv) =$
9            $MergeSideEffectsParamsCom((EnterPrivateMemory\ mc),$
10               $paramComs, sideEffect))$
```

Figure 3.19: *TranslateExpression* function that illustrates the translation of enterPrivate-Memory method calls.

expression that represent the behaviour of the method call. If the left expression of the method call is not another embedded method call or a sequence, the *sideEffect* is just c, as shown on lines 17-21.

For example, consider the statement `b = getA(x = y).getB();`. This statement has two nested method calls, and an embedded side effect in the parameter of the first call. This is translated into the following *SCJ-mSafe* code.

```
x = y;
A var0;
getA(x, var0);
var0.getB(b);
```

The result of applying *GetMethodCallReturnDec* to the embedded method call `getA` is the sequence command (*Seq*) that contains the new declaration of `var0` and method call `getA(x, var0)`, and the expression `var0` that is used as the left expression for the second method call `getB(b)`.

The translation of method calls needs to identify which method is being called, and, if appropriate, returns the corresponding *SCJ-mSafe* command dedicated to the method call. For example, if the method call is to the `enterPrivateMemory` method, the result of the translation is the *SCJ-mSafe* command *EnterPrivateMemory*; this is shown in Figure 3.19.

When a call to `enterPrivateMemory` is identified, the `run` method of the runnable object passed as the first argument to the method is determined by the *FindMethods* function on line 4. The *FindMethods* function takes an expression that identifies the type of the left-hand side of the method call, the name of the method being called, the arguments passed to the method, and the translation environment. All method signatures in the translation environment that match the criteria are returned and recorded in *mc.methods*.

For example, if the call was `a.run()`, the *FindMethods* function identifies methods called `run` in the class type of expression `a`, with no arguments. The *FindMethods* function also identifies all possible methods in classes that inherit from the class, as dynamic binding cannot be resolved precisely with static checking techniques, and so all possible methods are considered; this is why each method signature has a list of class descendants.

The translation of regular methods that are not a specific part of the SCJ paradigm is similar to that presented above. The main distinction is the lack of a special command in *SCJ-mSafe* to represent the method; instead, a regular *MethodCall* command is used. In the event of a method call returning a value or object, an additional result argument is added to the method call. If the overall statement being analysed is an assignment, it is the left-hand side of the assignment; if it is embedded in some other statement, a new variable is introduced to specifically hold the result of the method call.

ExtractExpression The second function used to translate expressions is the *ExtractExpression* function, which ignores any embedded commands inside expressions and simply defines the result of the expression. It is used by *TranslateExpression* to extract the meaning of expressions whilst ignoring side effects. It takes an SCJ expression and returns an *SCJ-mSafe* expression. Figure 3.20 shows an extract of the function, and specifies how the results of assignments in SCJ are translated into *SCJ-mSafe* expressions.

The domain of *ExtractExpression* is the subset of well-typed SCJ expressions, as shown on line 2. For all expressions in its domain, the *SCJ-mSafe* expression is extracted based on the type of the input expression. For example, when *ExtractExpression* is applied to an assignment `a = b`, the result is defined as the application of *ExtractExpression* to the left-hand side of the assignment (line 7), which is `a` in this case. The right-hand side of the assignment is ignored, because *ExtractExpression* is not interested in the actual assignment itself, only the result of the assignment, which is the left-hand side. As the left-hand side (`a`) is an identifier, the result is the *SCJ-mSafe* expression *ID id* (line 12), which is an *SCJ-mSafe* identifier; the specific identifier *id* is a variable whose name is equal to the SCJ identifier being analysed (`a`).

Figure 3.21 shows additional cases in the *ExtractExpression* specification that define

```
1  $ExtractExpression : SCJExpression \pfun Expr$
   ----------------
2  $\mathrm{dom}\, ExtractExpression \subset WellTypedExprs$
3      $\land \forall scjExpr : \mathrm{dom}\, ExtractExpression \bullet$
4          ...
5          $\lor (\exists e1, e2 : SCJExpression$
6              $\mid scjExpr = assignment(e1, e2)$
7              $\bullet ExtractExpression\ scjExpr = ExtractExpression\ e1)$
8          ...
9          $\lor (\exists name : Name;\ id : Identifier \mid$
10             $scjExpr = identifier\ name$
11             $\land id = VariableName\ name$
12             $\bullet ExtractExpression\ scjExpr = ID\ id)$
```

Figure 3.20: *ExtractExpression* function illustrating how *SCJ-mSafe* expressions are extracted from SCJ assignments.

```
1  ...
2  $\lor (\exists e1, e2 : SCJExpression$
3      $\mid scjExpr = binary(e1, e2)$
4      $\bullet ExtractExpression\ scjExpr = Val)$
5  $\lor (\exists e1 : SCJExpression;\ name : Name;\ fa, fa2 : FieldAccess;$
6      $iden : Identifier;\ v : Variable$
7          $\mid scjExpr = memberSelect(e1, name)$
8          $\land v.name = name$
9          $\bullet (\mathbf{let}\ lhs == ExtractExpression\ e1$
10             $\bullet (lhs = ID\ iden$
11                 $\land fa = \langle iden \rangle \frown \langle var\ v \rangle$
12                 $\land ExtractExpression\ scjExpr = FA\ fa$
13             $\lor lhs = FA\ fa2$
14                 $\land fa = fa2 \frown \langle var\ v \rangle$
15                 $\land ExtractExpression\ scjExpr = FA\ fa)))$
16 $\lor (\exists e1 : SCJExpression;\ type : TypeElement$
17     $\mid scjExpr = instanceOf(e1, type)$
18     $\bullet ExtractExpression\ scjExpr = OtherExpr)$
```

Figure 3.21: *ExtractExpression* function illustrating binary expressions, field accesses, and instance-of comparisons.

the behaviour for binary expressions (lines 2-4), field accesses (lines(5-15), and instance-of comparisons (lines 16-18).

The result of analysing binary expressions in *ExtractExpression* is simply a value (*Val*), as defined on line 4. This is because the translation abstracts away from operators on primitive types, and the actual result is not important; it is sufficient for the analysis to know that the result of the expression is some value.

Member-select expressions in SCJ are translated to field accesses (*FA*) in *SCJ-mSafe*. Member selects have two components, the left-hand expression ($e1$), and the name of the variable being identified (*name*). The translation uses *ExtractExpression* again on the left-hand side of the member select (line 9) and records it in the variable *lhs*, and then combines it with the variable being selected in one sequence of identifiers, which is the definition of a field access in *SCJ-mSafe*. If the result of applying *ExtractExpression* to the left-hand side of the member select is a simple identifier, the result is a sequence containing the identifier followed the variable, as shown on lines 11 and 12. If the left-hand side is a field access, the result is the variable concatenated onto the end of the field access, as shown on lines 14 and 15.

Instance-of comparisons in SCJ are translated to the *SCJ-mSafe* expression *OtherExpr*, which identifies expressions that are not important to the translation. The analysis does not take into account the result of the instance-of comparisons, therefore it is not translated.

3.5.2 Translating commands

The translation of commands is simpler than expressions; Figure 3.22 shows an extract from the *TranslateCommand* function specification, and illustrates how for loops and if statements are translated into *SCJ-mSafe*.

The *TranslateCommand* function takes an SCJ command, an SCJ program, and a translation environment, and returns an *SCJ-mSafe* command. All SCJ commands in the domain of the function are in the set *WellTypedComs*, which defines all well-typed commands in SCJ.

For loops in *SCJ-mSafe* are very similar to those in SCJ; the main difference is the extraction of any side effects in the conditional expression. When a for loop in SCJ is identified (line 10), the resulting *SCJ-mSafe* command is a sequence of extracted side effects (line 13) followed by the *SCJ-mSafe* for command *For* (line 14). The *SimplifyCommandPair* function on line 12 is used to simplify the sequence of commands if there are no side effects in the expression. The translation of the initialisation command, main body of the loop,

```
1  $TranslateCommand : SCJCommand \times SCJmSafeProgram$
2        $\times\ TranslationEnv \nrightarrow Com$
   ─────────────
3  $\forall scjCom : SCJCommand;\ program : SCJmSafeProgram;$
4        $transEnv : TranslationEnv$
5     $\mid (scjCom, program, transEnv) \in \operatorname{dom} TranslateCommand$
6     $\land\ scjCom \in WellTypedComs$
7     $\bullet$
8        $\ldots$
9        $\lor\ (\exists c1, c2, c3 : SCJCommand;\ e1 : SCJExpression$
10          $\mid scjCom = for(c1, e1, c2, c3)$
11          $\bullet\ TranslateCommand(scjCom, program, transEnv) =$
12            $SimplifyCommandPair($
13               $(TranslateExpression(e1, transEnv)),$
14               $(For((TranslateCommand(c1, program, transEnv)),$
15                  $(ExtractExpression\ e1),$
16                  $(TranslateCommand(c2, program, transEnv)),$
17                  $(TranslateCommand(c3, program, transEnv))))))$
18       $\lor\ (\exists e1 : SCJExpression;\ c1, c2 : SCJCommand$
19          $\mid scjCom = if(e1, c1, c2)$
20          $\bullet\ TranslateCommand(scjCom, program, transEnv) =$
21            $SimplifyCommandPair($
22               $(TranslateExpression(e1, transEnv)),$
23               $(If((ExtractExpression\ e1),$
24                  $(TranslateCommand(c1, program, transEnv)),$
25                  $(TranslateCommand(c2, program, transEnv))))))$
```

Figure 3.22: *TranslateCommand* function illustrating the translation of for loops and if statements.

and the iteration command is specified by the *TranslateCommand* function, as shown on lines 14-17.

Similarly, the translation of if statements extracts any side effects from the expression, and creates a sequence before the *SCJ-mSafe* command *If*. For example, consider the if command `if((x = y) > 0)`; the side effect `x = y` is extracted and performed before the *If* command. The expression used as the conditional statement is specified using the *ExtractExpression* function on line 23, as its side effects have been removed with the *TranslateExpression* function on line 22. The true and false branches of the conditional are specified using the *TranslateCommand* function on lines 24 and 25 respectively.

One of the most complex commands to translate is the `return` statement inside a

$$
\begin{array}{l}
\ldots \\
\lor (\exists\, e1 : SCJExpression;\ lexpr : Expr;\ v : Variable \\
\quad \mid scjCom = return\ e1 \\
\quad \land v.name = Result \\
\quad \land v.varType.resultVar = True \\
\quad \bullet (e1 \neq null \\
\qquad \land ((\exists\, te : TypeElement;\ args : \mathrm{seq}\, SCJExpression;\ nI : newInstance \\
\qquad\quad \mid e1 = newArray(te, args) \\
\qquad\quad \ldots \\
\qquad \lor (\exists\, args : \mathrm{seq}\, SCJExpression;\ nI : newInstance;\ name : Name \\
\qquad\quad \mid e1 = newClass(name, args) \\
\qquad\quad \ldots \\
\qquad \lor (\exists\, le : SCJExpression;\ args : \mathrm{seq}\, SCJExpression;\ c : Com \\
\qquad\quad \mid e1 = methodInvocation(le, args) \\
\qquad\quad \ldots \\
\quad \lor e1 = null \\
\qquad \land TranslateCommand(scjCom, program, transEnv) = Skip))
\end{array}
$$

Figure 3.23: *TranslateCommand* function illustrating the return statement.

method. The translation is complex as the SCJ expression returned could be a simple value, or a more complex method call or instantiation, for example. If the return statement has an argument, that is, a value or object is being returned, the result of the translation is the assignment of the expression being returned to the result parameter of the enclosing method. If no expression is returned, the resulting command in *SCJ-mSafe* is *Skip*. Figure 3.23 shows the outline of the specification of the *TranslateCommand* function for `return` statements, which is similar to the translation of assignment statements in the *TranslateExpression* shown in Figure 3.17, as it distinguishes the type expression being returned.

3.6 Final considerations

This chapter has presented the *SCJ-mSafe* language and its corresponding formal model in Z. In addition a formal model of SCJ has made it possible to produce a formalisation of the strategy to translate programs from SCJ to *SCJ-mSafe*. The full formalisation of SCJ, *SCJ-mSafe*, and the translation strategy can be found in Appendices B, C, and D respectively.

It will not be possible to prove that the translation from SCJ to *SCJ-mSafe* is correct, in the sense that it preserves the properties of the original program. It does not: it is an abstraction. It is, however, possible to demonstrate that this formalisation is adequate for the detection of memory-safety violations in the original SCJ program by analysing the corresponding *SCJ-mSafe* abstraction that has been defined here. The definition of SCJ-mSafe and the translation strategy using a formal language, namely Z, further supports the possibility of proof of soundness of the technique.

The translation has been automated, as described in Chapter 5; the implementation and testing of the tool validates the formalisation defined here. The validation process uncovered errors with the formalisation that were addressed as part of the ongoing development and testing process; this is discussed further in Section 5.4.

The next chapter describes the checking technique to identify possible memory-safety violations in *SCJ-mSafe* programs that we obtain after the translation.

Chapter 4

Modelling and checking memory configurations

In this chapter, the components required to perform memory-safety analysis on a translated *SCJ-mSafe* program are described. Section 4.1 gives an overview of the technique and demonstrates how it is possible to check for memory-safety violations using a simple example. Section 4.2 describes an environment that holds information about reference variables and their corresponding objects. Section 4.3 gives a description of method properties, which are used to capture the behaviour of methods independently of the calling context. Section 4.4 describes how the environment is updated throughout the analysis of the program, whilst Section 4.5 describes how method properties are calculated from the *SCJ-mSafe* program. Section 4.6 defines the memory-safety inference rules that support the proof that an *SCJ-mSafe* program is memory safe using an environment and set of method properties. Section 4.7 describes how possible memory-safety violations are detected inside method properties, and Section 4.8 defines the inference rules for method properties. Finally, Section 4.9 summarises the chapter and makes some final considerations.

4.1 Introduction

This section gives an overview of the memory-safety checking technique described throughout the remainder of this chapter. The purpose of this section is to demonstrate the steps undertaken in the overall technique before the technical material is presented. An SCJ program that represents a simple list protocol (as described in [12]) is described along with the *SCJ-mSafe* translation, the calculated method properties, and the environment used throughout the checking procedure.

This example does not contain any memory-safety violations, however, the worked example presented here gives an indication as to how errors can be detected.

4.1.1 Analysing static variables and the safelet

Figure 4.1 shows the first part of the *SCJ-mSafe* translation of the example SCJ program, which includes the static variables of the program (declared in a `Handler1` class) and the safelet. As explained in the previous chapter, static declarations and initialisations are separated into the `static` and `sInit` components respectively. In this example, the static variable is of a primitive type, and it is initialised with some value `Val`. The overall order in which the static variables and their corresponding initializers are defined is not important, and the order in which they are analysed does not affect the ability to detect potential memory-safety violations.

Static variables in the environment are not recorded as fields of an object. They are instead global reference variables at the top-level.

The translation of the safelet is below the static variables on lines 9-18, and demonstrates that each component is empty except for the `getSequencer` method on line 14. This is because there are no fields and no constructor in the safelet. The `getSequencer` method declares a new variable called `sequencer` and initialises it with a new `MainMissionSequencer` object.

When analysing *SCJ-mSafe* programs, method properties for all non-paradigm methods are generated automatically; this includes the constructors of classes. An environment is also used to record information about aliasing and memory areas of objects; this environment is maintained throughout the analysis, as will be demonstrated with this example.

Before the checking phase is reached, method properties are generated for each method in the *SCJ-mSafe* program. These method properties are similar to the environment in the sense that they record information about the effects of executing a method. The key difference is that method properties are independent of the execution, therefore it is not possible to use explicit reference contexts, for example. Instead, meta-reference contexts are defined, which enable the description of behaviour in an abstract way.

The first expressions added to the environment are the static variables of a program; in this case, the `IN_DATA_REGISTER_ADDRESS` variable. The resulting environment is as

```
1  program {
2    static {
3      long IN_DATA_REGISTER_ADDRESS;
4    }
5    sInit {
6      IN_DATA_REGISTER_ADDRESS = Val;
7    }
8
9    safelet {
10     fields { }
11     init { }
12     constr () { }
13     initializeApplication { }
14     getSequencer {
15       MainMissionSequencer sequencer;
16       NewInstance(sequencer, Current, MainMissionSequencer,
            ());
17       Result = sequencer;
18     }
19   }
```

Figure 4.1: Simple protocol example - *SCJ-mSafe* safelet

follows:

$$
\begin{array}{l}
\{\ \ \mathit{IN\_DATA\_REGISTER\_ADDRESS} \mapsto \mathit{IN\_DATA\_REGISTER\_ADDRESS} \\
\}\ \mapsto\ \{ \\
\quad \mathit{IN\_DATA\_REGISTER\_ADDRESS} \mapsto \{\mathit{Prim}\} \\
\}
\end{array}
$$

The environment above records that the `IN_DATA_REGISTER_ADDRESS` expression exists and maps to itself (because every expression is aliased with itself), and that the reference context in which it resides is the *Prim* context, which represents primitive values. The *Prim* reference context is defined specially because primitive values cannot give rise to unsafety.

The `initializeApplication` method of the safelet is analysed next, as this is the first execution point of the paradigm after the static variables have been initialised. The `initializeApplication` method has no behaviour in this example, therefore the environment is not changed. The `getSequencer` method is analysed next, which declares and instantiates a new mission sequencer. After the declaration of the variable `sequencer` on line 15 in Figure 4.1, the environment is as follows.

$$\begin{array}{l}\{\ \ \mathit{IN\_DATA\_REGISTER\_ADDRESS} \mapsto \mathit{IN\_DATA\_REGISTER\_ADDRESS},\\ \quad \mathit{sequencer} \mapsto \mathit{sequencer}\\ \}\ \mapsto\ \{\\ \quad \mathit{IN\_DATA\_REGISTER\_ADDRESS} \mapsto \{\mathit{Prim}\}, \mathit{sequencer} \mapsto \{\}\\ \}\end{array}$$

At this point, the sequencer variable has been declared so it is added to the environment, but it has not been instantiated, so an empty set of possibilities is recorded for the reference context in which it resides, because currently it does not reference an object. At the point of instantiation on line 16 of Figure 4.1, the set of reference contexts for `sequencer` are updated to include *IMem*, to record that the new object has been created in immortal memory, and the method properties for the appropriate constructor of the `MainMissionSequencer` class are applied to the environment.

The mission sequencer class is shown in Figure 4.2; the method properties for the constructor of the `MainMissionSequencer` class are shown below. The method properties for constructors of classes record the declaration of all of the fields of the class and their respective initialisation commands.

$$\{\mathit{mission\_done} \mapsto \mathit{mission\_done}\} \mapsto \{\mathit{mission\_done} \mapsto \{\mathit{Rcs}\{\mathit{Prim}\}\}\}$$

The `mission_done` field of the mission sequencer is of a primitive type, and therefore the method properties above show that the result of executing the constructor will add a mapping from `mission_done` to itself in the set of aliases, associated with a mapping from `mission_done` to the specific meta-reference context $\mathit{Rcs}\{\mathit{Prim}\}$, which identifies primitive types.

The result of creating the new `MainMissionSequencer` in the `Current` reference context (as shown on line 16) and the application of the method properties above, produces the following environment.

$$\begin{array}{l}\{\ \ \mathit{IN\_DATA\_REGISTER\_ADDRESS} \mapsto \mathit{IN\_DATA\_REGISTER\_ADDRESS},\\ \quad \mathit{sequencer} \mapsto \mathit{sequencer},\\ \quad \mathit{sequencer.mission\_done} \mapsto \mathit{sequencer.mission\_done}\\ \}\ \mapsto\ \{\\ \quad \mathit{IN\_DATA\_REGISTER\_ADDRESS} \mapsto \{\mathit{Prim}\},\\ \quad \mathit{sequencer} \mapsto \{\mathit{IMem}\}, \mathit{sequencer.mission\_done} \mapsto \{\mathit{Prim}\}\\ \}\end{array}$$

The `sequencer` expression now references an object that resides in the immortal memory

```
missionSeq {
  fields {
    boolean mission_done;
  }
  init { }
  constr () {
    ...
    mission_done = Val;
  }

  getNextMission {
    if (mission_done) {
      mission_done = Val;
      MainMission mission;
      NewInstance(mission, Current, MainMission, ());
      Result = mission;
    } else {
      Result = null;
    }
  }
}
```

Figure 4.2: Simple protocol example - *SCJ-mSafe* mission sequencer

area (*IMem*), which is the reference context in which the `getSequencer` method is analysed. The `mission_done` field of the newly instantiated mission sequencer is referenced as *sequencer.mission_done* in the environment to ensure expressions with the same name in different scopes are appropriately distinguished.

During the analysis, a notion of the current expression is defined, which captures the expression of the current execution point. For example, when applying the method properties of the mission sequencer constructor to the environment, the current expression is *sequencer*, as this is the current execution point of the analysis.

The environment here is memory safe because there are no references that may introduce a memory-safety violation. For example, the static variable *IN_DATA_REGISTER_ADDRESS* is pointing to the *Prim* reference context, which is always safe, and the *mission_done* field of the mission sequencer *sequencer* also points to the *Prim* reference context. The *sequencer* variable is a local variable to the mission sequencer and is, therefore, checked against the reference context in which it was defined, which is the immortal memory area; as the object referenced by *sequencer* also resides in immortal memory then no memory violation may occur.

```
mission MainMission {
  fields {
    List list;
  }
  init { }
  constr () { }
  initialize {
    NewInstance(list, Current, List, ());
    Handler1 handler1;
    ...
    NewInstance(handler1, Current, Handler1, (list, ...));
    handler1.register();
    Handler2 handler2;
    ...
    NewInstance(handler2, Current, Handler2, (list, ...));
    handler2.register();
  }
  cleanUp {
    Skip;
  }
}
```

Figure 4.3: Simple protocol example - *SCJ-mSafe* mission

4.1.2 Analysing the mission sequencer

After the `getSequencer` method has been analysed, the `getNextMission` method of the mission sequencer is analysed; the mission sequencer is shown in Figure 4.2. The `getNextMission` method includes the declaration and instantiation of the variable `mission`, which references the only mission object in the program. The `getNextMission` method is executed in the mission memory area ($MMem$), therefore the `Current` reference context in which the `MainMission` class is instantiated on line 15 of Figure 4.2 is the mission memory area. The method properties of the `MainMission` constructor are shown below.

$$\{list \mapsto list\} \mapsto \{list \mapsto \{\}\}$$

The method properties above show that when a new `MainMission` object is instantiated, the field `list` is added to the environment, however it is not instantiated, therefore the set of reference contexts in which it may reside is empty. The resulting environment after the `getNextMission` method has been analysed is shown below.

$$
\begin{array}{l}
\{\ \ IN\_DATA\_REGISTER\_ADDRESS \mapsto IN\_DATA\_REGISTER\_ADDRESS, \\
\quad sequencer \mapsto sequencer, \\
\quad sequencer.mission\_done \mapsto sequencer.mission\_done, \\
\quad sequencer.mission \mapsto sequencer.mission, \\
\quad sequencer.mission.list \mapsto sequencer.mission.list \\
\}\ \mapsto\ \{ \\
\quad IN\_DATA\_REGISTER\_ADDRESS \mapsto \{Prim\}, \\
\quad sequencer \mapsto \{IMem\}, \\
\quad sequencer.mission\_done \mapsto \{Prim\}, \\
\quad sequencer.mission \mapsto \{MMem\}, \\
\quad sequencer.mission.list \mapsto \{\} \\
\}
\end{array}
$$

At this point the environment is still safe. As before, the expressions that point to the *Prim* context cannot introduce memory-safety violations; the *sequencer* expression still points to the immortal memory as before, and the *sequencer.mission* expression is a local variable declared in the `getNextMission` method which executed in the mission memory area. Therefore because the object referenced by *sequencer.mission* also resides in mission memory, there can be no error.

4.1.3 Analysing the mission

After the `getNextMission` method in the mission sequencer is analysed, the missions of the program are analysed. In this example, there is only one mission shown in Figure 4.3. Its instance is referenced by the expression *sequencer.mission*. If a program contains more than one mission, the order in which they are analysed does not matter, as will be explained later.

The `initialize` method of the mission on lines 7-17 is executed first. It instantiates the `list` variable with a new `List` object in the current memory area (which is the mission memory area) before creating a new instance of the two handlers in the program. The variables `handler1` and `handler2` reference instances of the `Handler1` and `Handler2` classes respectively, both of which are also instantiated in the current memory area.

The instantiation of the `List` class on line 8 creates a new object of type `List` referenced by `list` and also calls the constructor of the `List` class. The *SCJ-mSafe* translation of the `List` class is shown in Figure 4.4. The method properties of the constructor for the `List` class are shown below.

$$
\begin{array}{l}
\{\ \ val \mapsto val,\\
\quad next \mapsto next,\\
\quad empty \mapsto empty\\
\}\ \mapsto\ \{\\
\quad val \mapsto \{Rcs\{Prim\}\},\\
\quad next \mapsto \{\},\\
\quad empty \mapsto \{Rcs\{Prim\}\}\\
\}
\end{array}
$$

The resulting environment after the `list` variable has been instantiated with a new object is shown below.

$$
\begin{array}{l}
\{\ \ IN\_DATA\_REGISTER\_ADDRESS \mapsto IN\_DATA\_REGISTER\_ADDRESS,\\
\quad sequencer \mapsto sequencer,\\
\quad sequencer.mission\_done \mapsto sequencer.mission\_done,\\
\quad sequencer.mission \mapsto sequencer.mission,\\
\quad sequencer.mission.list \mapsto sequencer.mission.list,\\
\quad sequencer.mission.list.val \mapsto sequencer.mission.list.val,\\
\quad sequencer.mission.list.next \mapsto sequencer.mission.list.next,\\
\quad sequencer.mission.list.empty \mapsto sequencer.mission.list.empty\\
\}\ \mapsto\ \{\\
\quad IN\_DATA\_REGISTER\_ADDRESS \mapsto \{Prim\},\\
\quad sequencer \mapsto \{IMem\},\\
\quad sequencer.mission\_done \mapsto \{Prim\},\\
\quad sequencer.mission \mapsto \{MMem\},\\
\quad sequencer.mission.list \mapsto \{MMem\},\\
\quad sequencer.mission.list.val \mapsto \{Prim\},\\
\quad sequencer.mission.list.next \mapsto \{\},\\
\quad sequencer.mission.list.empty \mapsto \{Prim\}\\
\}
\end{array}
$$

After the variable `list` has been instantiated with a new object, variables that reference the two handlers in the program are defined and instantiated. The *SCJ-mSafe* translation of the `Handler1` class is shown in Figure 4.6. The method properties for the constructor of the `Handler1` class are shown below.

$$
\begin{array}{l}
\{\ \ in\_data\_register \mapsto in\_data\_register, list \mapsto list, list \mapsto listArg\\
\}\ \mapsto\ \{\\
\quad in\_data\_register \mapsto \{\}, list \mapsto \{Erc\ listArg\},\\
\}
\end{array}
$$

```
class List {
  fields {
    int val;
    List next;
    boolean empty;
  }
  init { }
  constr () {
    next = null;
    empty = Val;
  }
  method append(value) {
    List node;
    node = this;
    while (node.empty) {
      node = node.next;
    }
    node.val = value;
    if (Val) {
      NewInstance(node.next, Current, List, ());
    } else {
      node.next.empty = Val;
    }
    node.empty = Val;
  }
  ...
}
```

Figure 4.4: Simple protocol example - *SCJ-mSafe* list class

The method properties of the constructor illustrate that two fields `in_data_register` and `list` are added to the environment, and that the `list` field is aliased with the `listArg` variable, which is a parameter of the constructor. The reference context of the `in_data_register` field is not known because it is assigned to point to an object that is created with an infrastructure method call, and the technique currently uses a stub reference implementation. The reference context of the list is defined as the expression meta-reference context of the `listArg` variable ($Erc\ listArg$). This is not known in advance, however it can be calculated when the method properties are applied to an environment to reflect a particular call to the method, at which point, the information about the allocation of the argument is known.

The method properties for the `Handler2` constructor are not shown here to simplify the example. The resulting environment after the `initialize` method of the mission has been analysed is shown in Figure 4.5.

When the expression $sequencer.mission.handler1.list$ is added to the environment, the set of reference contexts in which it resides is dependent on the argument

$$
\begin{aligned}
&\{\ \ldots\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1 \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1,\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{in\_data\_register}\\
&\qquad \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{in\_data\_register},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{val} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{val},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{next} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{next},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{empty} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{empty},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{list} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{list}.\mathit{val} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{val},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{list}.\mathit{next} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{next},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{list}.\mathit{empty} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{empty},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}2 \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}2\\
&\}\ \mapsto\ \{\\
&\quad \ldots\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1 \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{in\_data\_register} \mapsto \{\},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list} \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{val} \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{next} \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{empty} \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{sequencer}.\mathit{mission}.\mathit{handler}2 \mapsto \{\mathit{MMem}\}\\
&\}
\end{aligned}
$$

Figure 4.5: Simple protocol example - Environment after the mission's initialize method has been analysed

passed to the method ($\mathit{Erc}\ \mathit{listArg}$). The reference context of the argument ($\mathit{sequencer}.\mathit{mission}.\mathit{list}$) is the mission memory (MMem) in this example, therefore the mapping $\mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list} \mapsto \{\mathit{MMem}\}$ is added to the environment.

As the method properties of the handler constructor included the entry $\mathit{list} \mapsto \mathit{listArg}$, the handler field `list` is updated to point to the same object as the argument passed as a parameter to the constructor. In this case, the argument is the `list` field of the mission object, which is referenced by the expression $\mathit{sequencer}.\mathit{mission}.\mathit{list}$. Therefore the resulting environment includes the mapping $\mathit{sequencer}.\mathit{mission}.\mathit{list} \mapsto \mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}$, as they are now aliased. As a result of this aliasing, all fields of the `list` object referenced from the mission are now fields of the object referenced from the handler. Additionally, the set of expressions that reference the fields of the object ($\mathit{sequencer}.\mathit{mission}.\mathit{list}.\mathit{next}$ and $\mathit{sequencer}.\mathit{mission}.\mathit{handler}1.\mathit{list}.\mathit{next}$, for

```
 1  handler Handler1 {
 2    fields {
 3      RawInt in_data_register;
 4      List list;
 5    }
 6    init {
 7      in_data_register = ...;
 8    }
 9    constr (listArg, priority, period, storage) {
10      list = listArg;
11    }
12
13    handleEvent {
14      int value;
15      int var6;
16      in_data_register.get(var6);
17      value = var6;
18      MemoryArea mission_memory;
19      GetMemoryArea(mission_memory, this);
20      MissionMemoryEntry memEntry;
21      NewInstance(memEntry, Current, MissionMemoryEntry,
          (value));
22      ExecuteInAreaOf(Erc mission_memory, memEntry);
23    }
24  }
```

Figure 4.6: Simple protocol example - *SCJ-mSafe* handler 1

example) are also aliased.

The environment after the `initialize` method of the mission has been analysed is memory safe. This is because no fields or local variables reference objects that reside in lower memory areas.

4.1.4 Analysing handlers

After the `initialize` method of the mission has been analysed, the handlers associated with the current mission are analysed. The order in which handlers are analysed does not matter due to the way in which the technique handles concurrency. The *SCJ-mSafe* translation of the `Handler1` class is shown in Figure 4.6.

The behaviours of handlers are recorded in the environment by analysing the `handleEvent` methods, which are executed in the per-release memory area of the associated handler. The `handleEvent` method shown on line 13 of Figure 4.6 reads in a value from a data register, passes the value to a new instance of the `MissionMemoryEntry` class (shown in Figure 4.7), and executes the `run` method of the `MissionMemoryEntry` class inside the memory area of the object referenced by the `mission_memory` variable via

```
class MissionMemoryEntry {
  fields {
    int value;
  }
  init { }
  constr (val) {
    value = val;
  }
  method run() {
    list.insert(value);
  }
}
```

Figure 4.7: Simple protocol example - *SCJ-mSafe* mission memory entry class

the `ExecuteInAreaOf` command.

The `run` method of the `MissionMemoryEntry` class inserts the value that was passed as a parameter to the constructor into the list object of the handler using the `List` class' `insert` method. The `insert` method checks to see if the value is already stored in the list, and adds it using the `append` method if not. The `append` method is shown in Figure 4.4 and its method properties are shown below.

$$
\begin{array}{l}
\{\ \ node \mapsto node, \\
\quad node \mapsto this, \\
\quad node.val \mapsto node.val, \\
\quad node.val \mapsto value, \\
\quad node.next \mapsto node.next, \\
\quad node.next.next \mapsto node.next.next, \\
\} \ \mapsto \ \{ \\
\quad node \mapsto \{Erc\ this, Erc\ node.next\}, \\
\quad node.val \mapsto \{Erc\ value\}, \\
\quad node.next \mapsto \{Current\}, \\
\}
\end{array}
$$

In calculating the method properties for the `append` method, the `while` loop is analysed, which iteratively updates the local variable `node` to point to the last object in the list at run time. If, for example, there were four elements in the list, the local variable `node` would be aliased with the object referenced by `list.next.next.next`. When analysing loops, a fixed point can be used to capture the behaviour regardless of the number of iterations; however, in this case, a fixed point cannot be calculated as it is not possible to determine statically how many objects are stored in the list.

To overcome this problem, the technique uses a loop summary that is sufficient to

$$
\begin{aligned}
\{\ & sequencer.mission.handler1.list \mapsto sequencer.mission.handler1.list,\\
& sequencer.mission.handler1.list.val \mapsto sequencer.mission.handler1.list.val,\\
& sequencer.mission.handler1.list.next \mapsto sequencer.mission.handler1.list.next,\\
& sequencer.mission.handler1.list.empty \mapsto sequencer.mission.handler1.list.empty,\\
& sequencer.mission.handler1.value \mapsto sequencer.mission.handler1.value,\\
& sequencer.mission.handler1.memEntry\\
& \quad \mapsto sequencer.mission.handler1.memEntry,\\
& sequencer.mission.handler1.memEntry.value\\
& \quad \mapsto sequencer.mission.handler1.memEntry.value,\\
& sequencer.mission.handler1.list.node \mapsto sequencer.mission.handler1.list.node,\\
& sequencer.mission.handler1.list.node \mapsto sequencer.mission.handler1.list,\\
& sequencer.mission.handler1.list.node.val\\
& \quad \mapsto sequencer.mission.handler1.list.node.val,\\
& sequencer.mission.handler1.list.node.val\\
& \quad \mapsto sequencer.mission.handler1.list.val,\\
& sequencer.mission.handler1.list.node.next\\
& \quad \mapsto sequencer.mission.handler1.list.node.next,\\
& sequencer.mission.handler1.list.node.next\\
& \quad \mapsto sequencer.mission.handler1.list.next,\\
& sequencer.mission.handler1.list.node.next.next\\
& \quad \mapsto sequencer.mission.handler1.list.next.next,\\
& \ldots\\
\}\ & \mapsto\ \{\\
& sequencer.mission.handler1 \mapsto \{MMem\},\\
& sequencer.mission.handler1.list \mapsto \{MMem\},\\
& sequencer.mission.handler1.list.val \mapsto \{MMem\},\\
& sequencer.mission.handler1.list.next \mapsto \{MMem\},\\
& sequencer.mission.handler1.list.empty \mapsto \{MMem\},\\
& sequencer.mission.handler1.mission\_memory \mapsto \{MMem\},\\
& sequencer.mission.handler1.memEntry \mapsto \{PRMem\ handler1\},\\
& sequencer.mission.handler1.memEntry.value \mapsto \{Prim\},\\
& sequencer.mission.handler1.list.node \mapsto \{MMem\},\\
& sequencer.mission.handler1.list.node.val \mapsto \{Prim\},\\
& sequencer.mission.handler1.list.node.next \mapsto \{MMem\},\\
& sequencer.mission.handler1.list.node.next.next \mapsto \{\},\\
& \ldots\\
\}\ &
\end{aligned}
$$

Figure 4.8: Simple protocol example - Environment after Handler1 has been analysed

detect possible memory-safety violations. Loop summaries define the behaviour of the loop and are calculated in a single pass as will be explained in Section 4.4.

The method properties above state that after the method is executed, the variable `node` resides in either the reference context associated with the target object (`this`), or the reference context of the next element in the list (`node.next`). The field `node.val` resides in the reference context of the argument `value`, and the field `node.next` resides in the *Current* reference context as it is instantiated with a new object on line 20 of the `append` method.

The resulting environment after the `handleEvent` method of the `Handler1` class has been analysed is shown in Figure 4.8. The environment contains all of the local variables declared in the `handleEvent` method plus the method properties of the `append` method that is subsequently called by the `run` method of the object referenced by `memEntry`.

The environment is memory safe despite the fact that a new element of the list is instantiated and added whilst executing in the per-release memory area of the handler. This is because the `run` method, which is responsible for calling the subsequent methods that create the new instance of the list element, is executed in the same memory area in which the list object resides, which is the mission memory area. In other words, all elements of the list remain in the mission memory area, and no downward reference is created to a new list element created in the per-release memory area, which would have been the case had the `executeInAreaOf` method not been used.

After all handlers have been analysed, the `cleanUp` method of the mission is analysed. Subsequent missions are then analysed until there are no more missions to analyse. At this stage, the analysis is complete as there is no more user-defined code to be executed before the program terminates.

The remainder of this chapter explains the details of the analysis and checking technique outlined in this example. A formalisation of the checking technique is presented throughout, starting with a model of an environment that records information about the expressions in a program.

4.2 An environment for memory configurations

In order to check the memory-safety of an *SCJ-mSafe* program, information about the locations in which objects are allocated must be recorded. The memory area in which an object resides is represented as a reference context. Reference contexts capture the memory areas of SCJ plus an additional fictitious context called *Prim*, which represents primitive values. The formalisation of a type *RefCon* of reference contexts is shown below.

$$
\begin{aligned}
\mathit{RefCon} ::= {} & \mathit{Prim} \\
& \mid \mathit{IMem} \\
& \mid \mathit{MMem} \\
& \mid \mathit{PRMem}\langle\!\langle \mathit{Name} \rangle\!\rangle \\
& \mid \mathit{TPMem}\langle\!\langle \mathit{Name} \times \mathbb{N} \rangle\!\rangle \\
& \mid \mathit{TPMMem}\langle\!\langle \mathbb{N} \rangle\!\rangle
\end{aligned}
$$

As mentioned above, the *Prim* reference context is used for primitive values, *IMem* represents the immortal memory area, *MMem* the mission memory area, *PRMem* the per-release memory area of a specific handler, identified by the name of the handler, *TPMem* the temporary private memory areas of a handler, identified by the name of the handler and a natural number to capture nesting depth, and finally *TPMMem*, which represents temporary private mission memory areas, identified with a natural number to represent nesting depth, as used during the initialisation phase of a mission. Temporary private mission memory areas are identified only by a natural number to capture the nesting depth; since only one mission can execute at a single time, there is no need to identify the associated mission like with handlers.

An ordering can be defined on these reference contexts based on the memory hierarchy of SCJ, which allows us to check for possible memory-safety violations. This ordering is defined in Section 4.6.

The remainder of this section introduces an expression reference set, which records information about the reference contexts in which objects may reside, and an expression share relation, which records information about the aliasing of a program. Both of these elements make up the overall environment, which is also described. Finally, the method of handling concurrency is presented.

4.2.1 Expression reference sets

The environment is used to record the reference contexts of objects referenced by variables and fields. It is not always possible to determine precisely which reference context an object may reside using a static analysis technique. For example, consider the code excerpt below.

```
1 ...
2 Object o;
3 if (x > y) {
4     o = new Object();
5 } else {
6     MemoryArea memArea = MemoryArea.getMemoryArea(var);
7     o = memArea.newInstance(Object.class);
8 }
9 ...
```

In this simple example, the memory area in which the object referenced by variable `o` resides is dependent on the path of execution through the conditional statement. If `x` is greater than `y`, then `o` is instantiated in the current memory area (line 4), however, if `x` is not strictly greater than `y`, then `o` is instantiated in the memory area of the object referenced by `a` (lines 6-7).

In order to capture the necessary information to check memory safety, both the true and false branches of the conditional must be considered. If, as in this example, the resulting reference context of an object referenced by a left expression is different depending on the branch of execution, the set of all possible reference contexts must be recorded.

If, for example, the object referenced by some variable `var` resides in *IMem*, and the current allocation context is *MMem*, the object referenced by `o` may reside in either *IMem* or *MMem* as a result of the conditional statement. The corresponding entry in an expression reference set for the expression `o` would be as follows.

$$\{o \mapsto \{IMem, MMem\}\}$$

The definition of an expression reference set is captured by the type *ExprRefSet* below. It contains mappings from left expressions to sets of reference contexts.

$$ExprRefSet == LExpr \pfun \mathbb{P}\, RefCon$$

An expression reference set captures a worst-case model of the execution if the precise execution path cannot be determined, and can lead to different memory configurations.

4.2.2 Expression share relations

Another important aspect of programs that needs to be captured for the analysis is aliasing. If two reference variables point to the same object and a field of the object is updated through one of the reference variables, the change must also be reflected in the other.

Consider this very simple example below that illustrates the need to capture aliasing.

```
1  ...
2  o1 = new Object();
3  o2 = new Object()
4  a = o1;
5  o1.field = o2;
6  ...
7  a.field.var = ...
8  ...
```

In the example, the assignment of `o1` to `a` on line 4 means that the following assignment on line 5 to a field of the referenced object must be reflected in both `o1` and `a`. After line 5, `o1.field` references the same object as that referenced by `o2`; however, as `a` is aliased with `o1`, the field `a.field` also references `o2`. This is important to capture as the assignment on line 7 illustrates: if the aliasing had not been captured, further changes to fields of `a` would not be recorded correctly and potential errors may be missed.

To capture aliasing, an expression share relation that is an element of the set *ExprShareRelation* below is used. These are relations between two left expressions.

$$ExprShareRelation == LExpr \leftrightarrow LExpr$$

The *ExprShareRelation* and *ExprRefSet* definitions are the two elements used in an environment. Together they record sufficient information about the left expressions in an *SCJ-mSafe* program to facilitate memory-safety checking.

4.2.3 The environment

The environment to capture the necessary information about an *SCJ-mSafe* program is defined as a pair. The first element of the pair is an expression share relation, and the second element is an expression reference set. The formalisation of the environment is shown below; a table showing a description of the Z notation used can be found in

Appendix A.

$$\begin{array}{l}
\mathit{Env} == \{\mathit{env} : \mathit{ExprShareRelation} \times \mathit{ExprRefSet} \\
\quad \mid \forall\, \mathit{rel}, \mathit{crel} : \mathit{ExprShareRelation};\ \mathit{ref} : \mathit{ExprRefSet} \\
\qquad \mid (\mathit{rel}, \mathit{ref}) = \mathit{env} \\
\qquad \wedge\ \mathit{crel} = \mathit{rel}^{*} \cup (\mathit{rel}^{*})^{\sim} \\
\qquad\quad \bullet\ \mathrm{dom}\, \mathit{crel} = \mathrm{dom}\, \mathit{ref} \\
\qquad\quad \wedge\ (\forall\, e_1, e_2 : \mathit{LExpr} \\
\qquad\qquad \mid e_1 \mapsto e_2 \in \mathit{crel} \\
\qquad\qquad\quad \bullet\ \mathit{ref}\ e_1 = \mathit{ref}\ e_2)\}
\end{array}$$

There are two invariants satisfied by an environment; the first of these states that the domain of the expression reference set is equal to the domain of the reflexive, symmetric, transitive closure of the expression share relation. The reflexive, symmetric, transitive closure of the expression share relation is taken because all three properties hold when describing aliasing. For example, it is always true that an expression is aliased with itself, therefore the relation is reflexive. It is true that if an expression `a` is aliased with another expression `b`, then `b` is also aliased with `a`, therefore it is symmetric. And finally, if `a` is aliased with `b`, and `b` is aliased with `c`, it is true that `a` is aliased with `c`, therefore the relation is transitive. The domain of the reference set is equal to this because the environment must record information about every expression.

The second invariant on environments states that all expressions related by the reflexive, symmetric, transitive closure of the expression share relation must have the same set of reference contexts in the expression reference set. More specifically, if two expressions reference the same object, that is, they are aliased, then the set of possible reference contexts in which the object resides must be the same for each expression. It does not make sense for two expressions that point to the same object to have two different sets of reference contexts.

During the checking phase, the environment is updated after every command; if the changes made introduce a possible memory-safety violation, the command is identified as a possible source of error. The way in which the SCJ constructs affect the environment is described in detail in Section 4.4.

4.2.4 Handling concurrency

In order to handle concurrency for Level 1 programs, the environment described above includes the notion of history. This history records the set of all possible aliases, and the

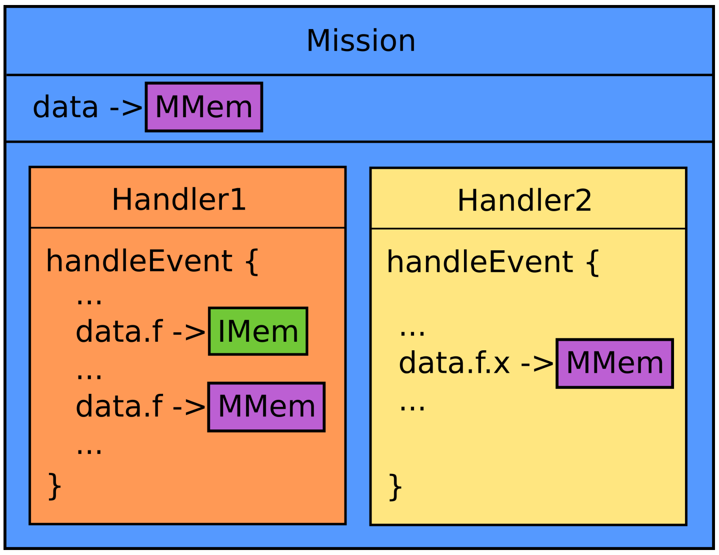


Figure 4.9: Possible memory-safety violation introduced through concurrency.

set of all possible reference contexts of expressions and objects throughout the analysis of a program. The changes made to the environment when it is updated are, therefore, non-destructive.

Figure 4.9 describes how concurrency can introduce possible memory-safety violations that would not be caught without preserving the history of aliases and reference contexts. The field of the mission object (`data`) is shared between the handlers of the mission. The `handleEvent` method of `Handler1` updates a field `f` of the shared object referenced by `data` to first point to an object in immortal memory, and then to an object in mission memory. Both of these operations are safe as the object referenced by `data` resides in mission memory.

The assignment in the `handleEvent` method of `Handler2` is also valid, as it assigns a field `x` of the object referenced by `data.f` to point to an object in mission memory. However, if, because of concurrency, the assignment in `Handler2` occurs in between the two assignments in `Handler1`, a possible memory-safety violation may arise. This is because the object `data` resides in mission memory, its field `data.f` resides in immortal memory, and the subsequent field `data.f.x` resides in mission memory, hence creating a downward reference from immortal memory to mission memory.

If `Handler1` is analysed without the notion of history, the resulting environment contains the mapping $\{data.f \mapsto \{MMem\}\}$. When `Handler2` is then analysed, the mapping $\{data.f.x \mapsto \{MMem\}\}$ is considered safe. If, however, the history of possible reference contexts is maintained, the resulting environment after analysing `Handler1` would contain the mapping $\{data.f \mapsto \{IMem, MMem\}\}$, as both reference contexts were possible at some stage during the execution. When `Handler2` is then analysed, the possible unsafe mapping from *IMem* to *MMem* is detected. If the handlers were analysed in the opposite order, the unsafe mapping would be detected when the field `data.f` is assigned to point to an object that resides in immortal memory, which is the first assignment in `Handler1`.

The history element of the environment also caters for the possibility of dynamic mission sequencing, and handler scheduling. More specifically, the order in which missions and handlers are analysed does not affect the analysis, as they are all analysed in the context of every other one.

The notion of history does, however, introduce the possibility of false information being recorded in the environment. This is because previous aliases and reference contexts of objects are not removed when they are updated. This does not make the technique unsound, but does increase the chances of false-negatives being raised. For example, consider the simple extract of code below.

```
1   Node n;
2   Node pt = new Node();
3   n = pt;
4   pt.next = new Node();
5   pt = pt.next;
```

In this example, a recursive data structure is created, and a new object is added on line 4. The head of the data structure is referenced by the variable `n`; the variable `pt` is updated to point to the last element in the structure. At the end of line 4, the resulting environment is as follows.

$$\begin{aligned}
&\{\ \ n \mapsto n, pt \mapsto pt, n \mapsto pt, pt.next \mapsto pt.next, n.next \mapsto n.next, n.next \mapsto pt.next,\\
&\quad n.next.next \mapsto n.next.next, pt.next.next \mapsto pt.next.next,\\
&\quad n.next.next \mapsto pt.next.next\\
&\}\ \mapsto\ \{\\
&\quad n \mapsto \{MMem\}, pt \mapsto \{MMem\}, n.next \mapsto \{MMem\}, pt.next \mapsto \{MMem\},\\
&\quad n.next.next \mapsto \{\}, pt.next.next \mapsto \{\}\\
&\}
\end{aligned}$$

The environment above shows that the expressions `n` and `pt` are aliased, and therefore all subsequent fields are also aliased. Assuming that this code executes in the mission memory area, it also illustrates that `n`, `pt`, `n.next`, and `pt.next` all reside in the mission memory area (*MMem*).

At this point in the analysis, the environment is correct, however, after line 5 has been analysed, the environment contains historic and current information, as shown below.

$$
\begin{array}{l}
\{\ \ n \mapsto n, n \mapsto pt, n \mapsto pt.next, pt \mapsto pt, pt \mapsto n.next, pt.next \mapsto pt.next, \\
\quad n.next \mapsto n.next, n.next \mapsto pt.next, pt.next.next \mapsto pt.next.next, \\
\quad n.next.next \mapsto n.next.next, pt.next.next \mapsto n.next.next \\
\}\ \mapsto\ \{ \\
\quad n \mapsto \{MMem\}, pt \mapsto \{MMem\}, n.next \mapsto \{MMem\}, pt.next \mapsto \{MMem\}, \\
\quad n.next.next \mapsto \{\}, pt.next.next \mapsto \{\} \\
\}
\end{array}
$$

The result of executing the command `pt = pt.next` is that the variable `pt` is updated to point to the next object in the data structure; all previous information about the aliases of `pt` is no longer current. During the analysis, which preserves history, the previous information about the aliases of `pt` is not discarded. For example, the alias $n \mapsto pt$ remains in the environment, despite the fact that `pt` is now actually aliased with `n.next`. The new alias $pt \mapsto n.next$ is added to the environment as expected, but historic information is maintained.

This historic information increases the possibility of false-negatives being raised during the analysis phase. On the other hand, it also facilitates the checking of recursive data structures, and allows recursive methods and loops to be analysed in a single iteration, which is a significant advantage to the technique, as will be described in Section 4.4.

4.3 Method properties

The environment described in Section 4.2 is used to record information about the expressions and references during the checking phase of the technique. Method properties record information about the effects of executing a method. It is not possible to use explicit reference contexts as in the environment, therefore meta-reference contexts (elements of *MetaRefCon*) are defined, which enable the description of behaviour in an abstract way.

$$
\begin{array}{rl}
MetaRefCon ::= & Rcs\langle\!\langle \mathbb{P}\ RefCon \rangle\!\rangle \\
| & Erc\langle\!\langle LExpr \rangle\!\rangle \\
| & Current \\
| & CurrentPrivate\langle\!\langle \mathbb{N} \rangle\!\rangle \\
| & CurrentPlus\langle\!\langle \mathbb{N} \rangle\!\rangle
\end{array}
$$

The first constructor in the definition above defines a specific reference context (*Rcs*), and is used when the reference context of an expression can be identified explicitly, regardless of the execution context. The best example of this scenario is primitive types, which always reside in the *Prim* reference context. The *Erc* meta-reference context is used to identify that the set of possible reference contexts of an expression is equal to that of another expression, or the Expression Reference Context (*Erc*) of that expression. For example, if `a` is assigned to `b`, the meta-reference context *Erc* can be used to describe the set of possible reference contexts of `b` in terms of `a`, without precise knowledge of the reference contexts associated with `a`.

The three remaining meta-reference contexts: *Current*, *CurrentPrivate*, and *CurrentPlus*, are used to describe expressions whose associated object resides in the current reference context, a more nested reference context, or an outer reference context, respectively. The *Current* context is the reference context in which the method is called. The *CurrentPrivate* context has a natural number associated with it, which defines the nesting level of temporary private memory areas that the object resides in. For example, if an object that resides in the *Current* context has a field that resides in a *CurrentPrivate* context, a possible memory-safety violation exists, because a downward reference occurs. The *CurrentPlus* context is the opposite of *CurrentPrivate*, and describes the number of levels up the scope an object is allocated in. For example, a reference from *Current* to *CurrentPlus* is safe, because the *CurrentPlus* context will outlive the *Current* one.

Consider the example method body shown below, which illustrates the need for the *Current*, *CurrentPrivate*, and *CurrentPlus* meta-reference contexts.

```
1  void myMethod() {
2      Object a = new Object();
3      MyRunnable myRun = new MyRunnable();
4      ManagedMemory.enterPrivateMemory(100, myRun);
5      ManagedMemory.executeInOuterArea(myRun);
6  }
7
8  class MyRunnable implements Runnable {
9      void run() {
10         Object b = new Object();
11     }
12 }
```

In this method body, the local variable `a` on line 2 references an object that resides in the *Current* meta-reference context, because it is instantiated in the memory area in which the method is called. The call to the `enterPrivateMemory` method on line 4 calls the `run` method of the `MyRunnable` class, which declares and instantiates the local variable `b` with a new object. As a temporary-private memory area has been entered, the object referenced by `b` resides in a memory area that is lower than the memory area in which the method is called, hence the object referenced by `b` resides in a *CurrentPrivate* meta-reference context.

Finally, the call to the `executeInOuterArea` method on line 5 executes the same `run` method of the `MyRunnable` class but in the next outer scope from the memory area in which the method was called. In this case, the local variable `b` references an object that resides in a higher memory area, therefore it is defined to reside in a *CurrentPlus* meta-reference context.

To capture which meta-reference contexts an expression of a method may reside in, a method reference set function (element of *MethodRefSet*) is used, which maps left expressions to a set of possible meta-reference contexts.

$$MethodRefSet == LExpr \nrightarrow \mathbb{P}\, MetaRefCon$$

This is very similar to the *ExprRefSet* in the environment, except that *MetaRefCon*s are used in the place of *RefCon*s. It is still important to capture the aliasing of the commands in a method, therefore, the method properties of a particular method are described with an expression share relation (*ExprShareRelation*) and a method reference set (*MethodRefSet*).

$$\begin{aligned}
&MethodProperties == \{properties : ExprShareRelation \times MethodRefSet \\
&\quad \mid \forall\, rel, crel : ExprShareRelation;\ ref : MethodRefSet \\
&\qquad \mid (rel, ref) = properties \\
&\qquad \land\ crel = rel^{*} \cup (rel^{*})^{\sim} \\
&\qquad\quad \bullet\ \mathrm{dom}\ crel = \mathrm{dom}\ ref \\
&\qquad\quad \land\ (\forall\, e_1, e_2 : LExpr \\
&\qquad\qquad \mid e_1 \mapsto e_2 \in crel \\
&\qquad\qquad\quad \bullet\ ref\ e_1 = ref\ e_2)\}
\end{aligned}$$

Similarly to the environment, there are two invariants satisfied by *MethodProperties*; the first states that the domain of the method reference set is equal to the domain of the reflexive, symmetric, transitive closure of the expression share relation.

The second invariant states that all expressions related by the reflexive, symmetric, transitive closure of the expression share relation must have the same set of meta-reference contexts in the method reference set. More specifically, if two expressions reference the same object, that is, they are aliased, then the set of possible meta-reference contexts in which the object they refer to resides must be the same for each expression.

$$
\begin{array}{l}
\{\ \ var13 \mapsto var13, \\
\quad var14 \mapsto var14, \\
\quad var15 \mapsto var15, \\
\quad var16 \mapsto var16, \\
\quad calibration \mapsto var16, \\
\quad var16 \mapsto calibration, \\
\quad wheel\_shaft \mapsto shaft, \\
\quad shaft \mapsto wheel\_shaft, \\
\quad \ldots \\
\}\ \mapsto\ \{ \\
\quad var13 \mapsto \{Current\}, \\
\quad var14 \mapsto \{Current\}, \\
\quad var15 \mapsto \{Current\}, \\
\quad var16 \mapsto \{Prim\}, \\
\quad calibration \mapsto \{Erc\ var16\}, \\
\quad wheel\_shaft \mapsto \{Erc\ shaft\}, \\
\quad \ldots \\
\}
\end{array}
$$

Figure 4.10: Extract of method properties for *SpeedMonitor* constructor in ACCS *SCJ-mSafe* program.

Figure 4.10 shows an extract from the method properties associated with the *SpeedMonitor* constructor (originally shown in Figure 3.2). The expression share relation captures the new local variables that are created, along with the alias, changes should the method be called. For example, the *shaft* expression is a parameter of the constructor, and is assigned to the local field *wheel_shaft* during the method's execution. The method reference set reflects this by including the entry $wheel\_shaft \mapsto \{Erc\ shaft\}$, which captures the fact that *wheel_shaft* points to an object that resides in the set of reference contexts associated with the parameter *shaft*, which is determined by the argument passed at the point of the method call.

The next section describes how the environment is updated throughout the analysis of a program; it also highlights how the method properties described here are used.

4.4 Updating the environment

In order to define the resulting environment after every construct in an *SCJ-mSafe* program, a series of *CalcEnv* functions are defined. These functions take an environment and an *SCJ-mSafe* component, and return an updated environment based on the analysis of that component.

The resulting environments calculated after specific *SCJ-mSafe* components have been analysed are used by other *CalcEnv* functions for components higher in the paradigm; for example, the resulting environment after a handler has executed is used by the containing mission. They are also used by the memory-safety inference rules as discussed in Section 4.6.

A full formalisation of the analysis can be found in Appendix E; the interesting aspects are presented here. A description of how the environment is updated for all *SCJ-mSafe* commands is included along with the handler, mission, mission sequencer, and safelet components.

4.4.1 Commands

The *CalcEnvCom* function is used to calculate the environment based on the execution of a specific command. The function takes an environment, a command, a left expression, a reference context, and an *SCJ-mSafe* program. The environment is the current environment, the command is the command to be analysed, the left expression is the current expression, which defines the expression of the current execution point, the reference context is the current default reference context, and the *SCJ-mSafe* program is the one currently being analysed . The result of the function is a new environment that has been updated based on the specific command.

$$CalcEnvCom : Env \times Com \times LExpr \times RefCon \times SCJmSafeProgram \rightarrow\!\!\!\!\!+ \; Env$$

$$\forall\, env : Env;\; c : Com;\; cexpr : LExpr;\; rc : RefCon;\; p : SCJmSafeProgram \bullet \ldots$$

The *CalcEnvCom* function is essentially defined by a case-analysis for all *SCJ-mSafe* commands that updates the environment accordingly. The update procedures for each command are described individually below.

Skip The first command in the case-analysis is the skip command.

$$c = Skip \land CalcEnvCom(env, c, cexpr, rc, p) = env$$

The skip command has no behaviour, and therefore the environment remains unchanged. The result of the *CalcEnvCom* function when the command c is *Skip* is the original environment *env*.

Declarations Declarations only add new information to the environment; no existing information is changed. The resulting environment includes a mapping from the new variable that is being declared to itself in the expression share relation, and a mapping to either the reference context set $\{Prim\}$ for primitive types, or the empty set ($\{\}$) otherwise, in the expression reference set.

For example, consider the declaration `int i;`, which is a primitive type. When applied to an empty environment, the resulting environment is

$$\{i \mapsto i\} \mapsto \{i \mapsto \{Prim\}\}$$

Alternatively, consider the declaration `Object o;`, which is not primitive. When applied to an empty environment, the resulting environment is

$$\{o \mapsto o\} \mapsto \{o \mapsto \{\}\}$$

The formalisation of the declaration in *CalcEnvCom* is as follows.

$$\begin{array}{l} \lor\ (\exists\, d : Dec \\ \quad \bullet\ c = Decl\ d \land CalcEnvCom(env, c, cexpr, rc, p) = AddDecToEnv(env, d)) \end{array}$$

The result of the *CalcEnvCom* function when the command c is a declaration $Decl\ d$ is determined by the *AddDecToEnv* function, which takes an environment and a declaration and updates the environment accordingly. The *AddDecToEnv* function is one of many specific-case functions used by *CalcEnvCom* that are explained individually.

$$\begin{array}{|l} AddDecToEnv : Env \times Dec \rightarrow\!\!\!\!\!+ Env \\ \hline \forall\, env : Env;\ d : Dec \\ \quad \bullet\ \exists\, rel : ExprShareRelation;\ ref : ExprRefSet \\ \qquad \mid env = (rel, ref) \\ \qquad \bullet\ AddDecToEnv(env, d) = \\ \qquad\quad ExprShareAdd((ID(var\ d.var)), (ID(var\ d.var)), rel) \\ \qquad\quad \mapsto ExprRefAdd((ID(var\ d.var)), (GetDecRefCon\ d), ref) \end{array}$$

The function definition states that for all environments *env* and declarations d, there exists an expression share relation *rel* and an expression reference set *ref* such that the environment is defined as (rel, ref). Moreover, the result of $AddDecToEnv(env, d)$ is a mapping from the updated expression share relation to the updated expression reference context. The expression share relation *rel* update is defined by the *ExprShareAdd* function, which takes two left expressions and an expression share relation, and returns an updated expression share relation. In the case of a declaration, the two expressions passed to *ExprShareAdd* are the same $(ID(var\ d.var)))$. This is because the new expression being declared is aliased with itself and nothing else at this point.

The expression reference set *ref* is updated with the *ExprRefAdd* function, which takes an expression, a set of reference contexts, and the expressions reference set to be updated. It returns an expression reference set with a new mapping from the left expression to the set of possible reference contexts. In the case of a declaration, the set of possible reference contexts is either $\{Prim\}$, if the type of the declaration is primitive, or empty $(\{\})$ otherwise, as no object has been created yet and therefore the expression does not reference anything. This set of possible reference contexts is determined by the *GetDecRefCon* function. The Z definitions of the functions *ExprShareAdd*, *ExprRefAdd*, *GetDecRefCon*, and all other omitted here can be found in Appendix E.

Sequence Both commands in a sequence command (*Seq*) are analysed using the *CalcEnvCom* function. First the environment is updated to reflect the effect of the first command, then with the second.

$$
\begin{array}{l}
\lor\ (\exists\, c1, c2 : Com \\
\quad \bullet\ c = Seq(c1, c2) \\
\quad \land\ CalcEnvCom(env, c, cexpr, rc, p) = \\
\qquad CalcEnvCom(\\
\qquad\quad (CalcEnvCom(env, c1, cexpr, rc, p)), \\
\qquad\quad c2, cexpr, rc, p))
\end{array}
$$

The first command in the sequence $c1$ is analysed in the environment *env* with a recursive call to *CalcEnvCom*. This gives an updated environment, which is used as the environment passed to the *CalcEnvCom* function once again to analyse the second command $c2$.

Assignments The assignment command (*Asgn*) is one of the two most important commands when updating the environment. The changes to the environment are made based on the type of the left and right expressions. The assignment cases described here are based on those described in [17].

Right expressions that are values have no impact on the environment because primitive types always reside in the *Prim* reference context; it does not matter what the specific value is. For all other assignments, the type of update is dependent on the left expression. For example, consider the assignment `x = 10;` if the previous environment is

$$\{x \mapsto x\} \mapsto \{x \mapsto \{Prim\}\}$$

irrespective of the right-hand side, the resulting environment will still be

$$\{x \mapsto x\} \mapsto \{x \mapsto \{Prim\}\}$$

If the left expression is a variable, a mapping from the left expression to the right expression is included in the environment share relation as long as the left expression is not a prefix of the right expression. For example, in the assignment `node = node.next`, the variable `node` is not aliased with `node.next` as a result of this assignment, therefore the mapping is not included in the share relation.

The mappings for fields of the right expression, which are now fields of the left expression are also added. For example, consider the assignment `a = b;` where the current environment is

$$\{a \mapsto a, b \mapsto b, b.x \mapsto b.x\} \mapsto \{a \mapsto \{\}, b \mapsto \{MMem\}, b.x \mapsto \{Prim\}\}$$

The resulting environment is

$$\{a \mapsto a, b \mapsto b, a \mapsto b, a.x \mapsto a.x, b.x \mapsto b.x, a.x \mapsto b.x\} \\ \mapsto \{a \mapsto \{MMem\}, b \mapsto \{MMem\}, a.x \mapsto \{Prim\}, b.x \mapsto \{Prim\}\}$$

If the left expression is a field access or array element, the changes to the environment described above are performed plus an additional change that updates all expressions already in the environment that are equal to the left expression; any changes to fields of the object are also applied to other expressions that reference it. For example, consider the assignment `a.f = x;` where the current environment is

$$\{a \mapsto a, b \mapsto b, a \mapsto b, a.f \mapsto a.f, b.f \mapsto b.f, a.f \mapsto p, b.f \mapsto p\} \\ \mapsto \{a \mapsto \{MMem\}, b \mapsto \{MMem\}, a.f \mapsto \{IMem\}, b.f \mapsto \{IMem\}\}$$

The resulting environment, assuming that `x` resides in the mission memory area, is

$$\{a \mapsto a, b \mapsto b, a \mapsto b, a.f \mapsto a.f, b.f \mapsto b.f, a.f \mapsto x, b.f \mapsto x, a.f \mapsto p, b.f \mapsto p\} \\ \mapsto \{a \mapsto \{MMem\}, b \mapsto \{MMem\}, a.f \mapsto \{MMem\}, b.f \mapsto \{MMem\}\}$$

Here both `a.f` and `b.f` are updated to point to `x`, because `a` and `b` reference the same object. The mappings $a.f \mapsto p$ and $b.f \mapsto p$ remain a part of the resulting environment because of the history element described previously.

Overall, the result of the *CalcEnvCom* function when the command c is an assignment is determined by the *CalcEnvAssignment* function.

$$\begin{array}{l} \lor\ (\exists\, le : LExpr;\ re : Expr \\ \quad \bullet\ c = Asgn(le, re) \land CalcEnvCom(env, c, cexpr, rc, p) = \\ \quad\quad CalcEnvAssignment(env, le, re, cexpr, rc, p)) \end{array}$$

The *CalcEnvAssignment* function takes an environment, a left expression, an expression, another left expression, a reference context, and an *SCJ-mSafe* program. The environment is the current environment being updated, the first left expression is the left-hand side of the assignment, the expression is the right-hand side of the assignment, the other left expression is the current expression, the reference context is the current default reference context, and the *SCJ-mSafe* program is the program currently being analysed. The result is an updated environment that takes into account the effects of the assignment. The *CalcEnvAssignment* function specification is defined in Figure 4.11.

If the right expression of the assignment is equal to *Val*, it is of a primitive type, and so the result of *CalcEnvAssignment* is the original environment, as no changes are made; this is specified on lines 5 and 6.

If the right expression is not *Val* (line 7), the left and right expressions (*le* and *re*) are used to define expressions *newle* and *newre*, which have the completed names of *le* and *re*. This is achieved with the *MergeExprs* function, which takes two expressions and the current *SCJ-mSafe* program and returns a new expression that is a combination of both input expressions. For example, the result of merging the current expression *sequencer.mission.handler* and *handlerField* is the expression *sequencer.mission.handler.handlerField*.

The completed left expression of the assignment is defined based on the current expression using the *MergeExprs* function on line 9. If the right expression (*re*) is equal to *This*, then the left expression is being assigned a reference to the current object, so the new right expression *newre* is equal to the current expression *cexpr* (line 10); otherwise it is defined using the *MergeExprs* function like the left expression (line 11).

If the left expression is a variable (line 12), three functions are applied to the environment. The first is *ExprShareAddEnv*, which takes the new left and right expressions and adds the mapping $newle \mapsto newre$ to the expressions share relation (line 16). The second

```
1  $CalcEnvAssignment : Env \times LExpr \times Expr \times LExpr$
2        $\times\ RefCon \times SCJmSafeProgram \nrightarrow Env$
   ─────────────
3  $\forall\, env : Env;\ le, cexpr : LExpr;\ re : Expr;\ rc : RefCon;$
4        $p : SCJmSafeProgram$
5    $\bullet\ re = Val$
6        $\land\ CalcEnvAssignment(env, le, re, cexpr, rc, p) = env$
7    $\lor\ re \neq Val$
8        $\land\ (\exists\, newle : LExpr;\ newre : Expr$
9          $\mid newle = MergeExprs(cexpr, le, p)$
10         $\land\ (re = This \land newre = cexpr$
11         $\lor\ re \neq This \land newre = MergeExprs(cexpr, re, p))$
12         $\bullet\ ((\exists\, v : Variable \mid le = ID(var\ v)$
13           $\bullet\ CalcEnvAssignment(env, le, re, cexpr, rc, p) =$
14             $AddAsgnFields(newle, newre,$
15               $(ExprRefAddEnvAsgn(newle, newre,$
16                 $(ExprShareAddEnv(newle, newre, env)))))$
17           $\lor\ (\exists\, fa : FieldAccess;\ ae : ArrayElement$
18             $\mid le = FA\,fa$
29             $\lor\ le = ID(arrayElement\ ae)$
20               $\bullet\ CalcEnvAssignment(env, le, re, cexpr, rc, p) =$
21                 $UpdateEqualExprs(newle, newre,$
22                   $(AddAsgnFields(newle, newre,$
23                     $(ExprRefAddEnvAsgn(newle, newre,$
24                       $(ExprShareAddEnv(newle, newre, env)))))))))$
```

Figure 4.11: *CalcEnvAssignment* function that updates the environment based on the assignment command.

is the *ExprRefAddEnvAsgn* function, which adds a mapping from the new left expression to the set of reference contexts associated with the new right expression to the expression reference set (line 15). Finally the third is the *AddAsgnFields* function, which adds any fields of the new right expression as fields of the new left expression (line 14). For example, if the field `b.x` exists, and `b` is assigned to `a`, the expression `a.x` is included in the environment as it is aliased with `b.x`.

If the left expression is a field access or array element (line 17), four functions are applied to the environment. The first three are the same as those described above (lines 22-24). The fourth function is the *UpdateEqualExprs* function, which updates all expressions in the environment that are aliased with the left expression (line 21). For example, if `a` is aliased with `b` in the environment, and `a.x` is updated to reference `y`, the mapping $a.x \mapsto y$

is updated in the environment; however, it is also important to update `b.x`, as this too has been changed. The *UpdateEqualExprs* function would ensure the mapping $b.x \mapsto y$ is also updated, for example.

NewInstance The new instance command (*NewInstance*) is the second of the two most interesting commands regarding the update of the environment, as it is a possible source of memory-safety violations. The resulting environment includes a mapping from the expression being instantiated to the set of possible reference contexts in which the new object may reside. The method properties of the corresponding object constructor are also applied.

For example, consider the command `NewInstance(o, Current, Object, ())`, which instantiates a new `Object` in the current reference context with no arguments and assigns a reference to it to the expression `o`. If the environment before the instantiation is

$$\{o \mapsto o\} \mapsto \{o \mapsto \{\}\}$$

the resulting environment is

$$\{o \mapsto o\} \mapsto \{o \mapsto \{MMem\}\}$$

assuming the mission memory is the current reference context.

$$\begin{array}{l} \lor\ (\exists\, nI : newInstance \\ \quad \bullet\ c = NewInstance\ nI \land CalcEnvCom(env, c, cexpr, rc, p) = \\ \qquad CalcEnvNewInstance(env, nI, cexpr, rc, p)) \end{array}$$

The result of the *CalcEnvCom* function in the case of a new instance command is the result of the *CalcEnvNewInstance* function, which takes the environment, new instance command, current expression, current reference context, and the *SCJ-mSafe* program as arguments. The specification of the *CalcEnvNewInstance* function is defined in Figure 4.12.

When analysing a new instance command, the left expression being instantiated with a new object must be updated based on the current expression. The new left expression (*newLe*) introduced on line 5 is defined as the result of merging the current expression (*cexpr*) with the left expression of the new instance command (*nI.le*), which is achieved with the *MergeExprs* function (line 8).

Similarly, if the meta-reference context of the new instantiation states that the new object resides in the reference context of another expression (*Erc*), the associated expression must also be updated with the *MergeExprs* function (lines 9 and 10). The updated

```
1  $CalcEnvNewInstance : Env \times newInstance \times LExpr \times RefCon$
2        $\times\ SCJmSafeProgram \nrightarrow Env$
   ---
3  $\forall\, env : Env;\ nI : newInstance;\ cexpr : LExpr;\ rc : RefCon;$
4        $p : SCJmSafeProgram$
5     $\bullet\ \exists\, newLe, e1 : LExpr;\ newMrc : MetaRefCon;\ constr : Method;$
6        $rel : ExprShareRelation;\ ref : ExprRefSet$
7        $\mid env = (rel, ref)$
8        $\land newLe = MergeExprs(cexpr, nI.le, p)$
9        $\land (nI.mrc = Erc\ e1$
10          $\land newMrc = Erc(MergeExprs(cexpr, e1, p))$
11       $\lor nI.mrc \neq Erc\ e1$
12          $\land newMrc = nI.mrc)$
13       $\land constr = GetConstr(nI.type.type, nI.args, p)$
14          $\bullet\ CalcEnvNewInstance(env, nI, cexpr, rc, p) =$
15             $ApplyPossibleMethods(constr, nI.args,$
16                $(rel \mapsto ExprRefUpdate(newLe,$
17                   $(RCsFromMRC(newMrc, rc, ref, cexpr)), ref)),$
18             $cexpr, nI.le, rc, p)$
```

Figure 4.12: *CalcEnvNewInstance* function that updates the environment based on new instantiations.

meta-reference context is defined by the *newMrc* component introduced on line 5. If the meta-reference context is not that of another expression (line 11), then *newMrc* is simply defined to be equal to the meta-reference context of the new instance command ($nI.mrc$) on line 12.

A result of instantiating a new object is that the constructor of the new object is called immediately after the object is created. The constructor (*constr*) is determined with the *GetConstr* function on line 13, which analyses all of the constructors defined in the *SCJ-mSafe* program and identifies the relevant one based on the type of the object being instantiated and the parameters passed. The overall *SCJ-mSafe* program is required in order to determine all possible constructors.

The overall result of the *CalcEnvNewInstance* function is the result of the *ApplyPossibleMethods* function, which takes a set of possible methods and applies the associated method properties of the methods to the environment. In this case, the set of possible methods is a singleton set containing only the relevant constructor. The *ApplyPossibleMethods* function also takes an environment as an argument, which in this case is defined as the existing environment updated with a mapping from the left expression to the possible reference contexts the new object is being instantiated in. The set of reference contexts is determined by the *RCsFromMRC* function, which analyses the meta-reference context of the new instance command (*newMrc*) and returns a set of reference contexts based on the current reference context *rc*.

If the meta-reference context is simply *Current*, the resulting set of reference contexts returned by *RCsFromMRC* is a singleton set containing the current reference context. If, however, the meta-reference context identifies that the method is to be executed in the reference context of another expression *Erc e*, then the resulting set of reference contexts is dependent on the set of reference contexts associated with the expression *e* in the environment.

For every method in the set of possible methods associated with a call (one in the case of a constructor), the *ApplyPossibleMethods* function updates the corresponding method properties based on the arguments passed to the method, and subsequently calls the *ApplyMethodProperties* function. The method properties associated with a method include the expressions that correspond to the parameters, not the actual arguments. The *UpdateMethodPropertiesArgs* function, which is omitted here, replaces the expressions that represent parameters in the method properties with the corresponding argument expressions. For example, if a method definition had the following signature

```
  void myMethod(A a, B b) { ... }
```

and a call to the method was made with the arguments `x.y`, and `z` (`myMethod(x.y, z);`), then all instances of `a` in the method properties are replaced with `x.y`, and all instances of `b` are replaced with `z`.

Having updated the method properties to be applied, the *ApplyMethodProperties* function is called, which takes the method to be applied, the current environment, the arguments passed to the method, the left expression of the target object, the current expression, the current reference context, and the *SCJ-mSafe* program being analysed. The result is an updated environment with the method properties of the particular method having been applied. The specification of the *ApplyMethodProperties* function is defined in Figure 4.13.

The method properties to be applied (*m.properties*) to the environment (*env*) are based on the current expression. For example, if the current expression is *sequencer.mission* and the method properties included the entry $a \mapsto b$, the updated method properties would

```
1  $ApplyMethodProperties : Method \times Env \times \operatorname{seq} Expr \times LExpr$
2        $\times\ LExpr \times RefCon \times SCJmSafeProgram \nrightarrow Env$
   ----------
3  $\forall\, m : Method;\ args : \operatorname{seq} Expr;\ env : Env;\ cexpr, lexpr : LExpr;$
4      $rc : RefCon;\ p : SCJmSafeProgram$
5    $\bullet\ \exists\, rel, methRel : ExprShareRelation;\ ref : ExprRefSet;$
6      $methRef : MethodRefSet$
7      $\mid env = (rel, ref)$
8      $\land\ m.properties = (methRel, methRef)$
9        $\bullet\ \mathbf{let}\ updatedShare == UpdateMethodPropertiesCExprShare($
10             $methRel, args, m.visibleFields, cexpr, lexpr, p);$
11           $updatedRef == UpdateMethodPropertiesCExprRef(methRef,$
12             $args, m.visibleFields, cexpr, lexpr, p)$
13           $\bullet\ ApplyMethodProperties(m, m.properties,$
14               $env, args, cexpr, lexpr, rc, p) =$
15             $UpdateEqualExprsSet(updatedShare,$
16               $(AddAsgnFieldsSet(updatedShare,$
17                 $(ExprShareAddSet(updatedShare, rel)$
18                   $\mapsto ExprRefUpdateSet($
19                     $(RefSetFromMethodRef(updatedRef,$
20                       $ref, rc, cexpr)), ref)))))$
```

Figure 4.13: *ApplyMethodProperties* function that updates the environment based on the execution of methods.

include the mapping $sequencer.mission.a \mapsto sequencer.mission.b$. This is defined with the *UpdateMethodPropertiesCExprShare* and *UpdateMethodPropertiesCExprRef* function calls on lines 9 and 11; a description of these functions is omitted here.

The method of calculating the result of the overall *ApplyMethodProperties* function is similar to that of the *CalcEnvAssignment* function. Firstly, the reference set and share relation, updated with the mappings in the method properties (lines 17 and 18), are defined using the *ExprRefUpdateSet* and *ExprShareAddSet* functions. Next the fields of references that have changed are defined with the *AddAsgnFieldsSet* function on line 16. Finally, any expressions that are aliased with expressions that have been changed are defined with the *UpdateEqualExprsSet* function on line 15.

If Both the true and false branches of *If* statements are analysed individually, and the results are merged; the resulting environment reflects both possible behaviours.

$$
\begin{array}{l}
\lor\ (\exists\, e : \mathit{Expr};\ \ c1, c2 : \mathit{Com} \\
\quad \bullet\ c = \mathit{If}(e, c1, c2) \\
\quad \land\ \mathit{CalcEnvCom}(\mathit{env}, c, \mathit{cexpr}, \mathit{rc}, p) = \\
\qquad \mathit{EnvJoin}((\mathit{CalcEnvCom}(\mathit{env}, c1, \mathit{cexpr}, \mathit{rc}, p)), \\
\qquad\qquad\qquad (\mathit{CalcEnvCom}(\mathit{env}, c2, \mathit{cexpr}, \mathit{rc}, p))))
\end{array}
$$

Both $c1$ and $c2$ are analysed in the same environment *env* as only one or the other is actually executed at run-time. The two resulting environments are merged using the *EnvJoin* function, which takes two environments and returns a single environment based on the information in each. For example, if one environment records that some variable resides in mission memory, but the other records that the same variable resides in immortal memory, the merged environment contains a single entry for the variable with both mission and immortal memory areas as possible reference contexts.

Switch The *Switch* command is analysed based on all possible cases of its execution. The resulting environment is a summary of the behaviour of all possible cases like in the *If* statement above.

Loops Commands such as the *For*, *While*, and *DoWhile* loops are analysed by calculating a loop summary, by analysing a single iteration of the loop.

$$
\begin{array}{l}
\lor\ (\exists\, c1, c2, c3 : \mathit{Com};\ \ \mathit{exp} : \mathit{Expr} \\
\quad \bullet\ c = \mathit{For}(c1, \mathit{exp}, c2, c3) \\
\quad \land\ \mathit{CalcEnvCom}(\mathit{env}, c, \mathit{cexpr}, \mathit{rc}, p) = \\
\qquad \mathit{CalcEnvCom}((\mathit{CalcEnvCom}(\mathit{env}, c1, \mathit{cexpr}, \mathit{rc}, p)), \\
\qquad\quad (\mathit{Seq}(c2, c3)), \mathit{cexpr}, \mathit{rc}, p))
\end{array}
$$

In the case above, the result of analysing the *For* command is a composition of analysing the initialisation command $c1$ in the original current environment *env* followed by the analysis of the body of the loop and the iteration command ($c2$ and $c3$) in the resulting environment from analysing $c1$.

It is sufficient to analyse loops only once because of the history element of the environment. If the code being analysed contains a command that has an effect on the environment, it is recorded in the loop summary regardless of the iteration. Further analysis of the loop body is not required to capture all possible execution paths of all iterations.

For example, consider the simple example shown below, which demonstrates how the elements of an array are assigned.

```
1  for (int i = 0; i < store.size(); i++) {
2    local[i] = store[i];
3  }
```

In this example, the assignment on line 2 is translated to `local[Val] = store[Val]` in *SCJ-mSafe*, as the precise index of the array is abstracted away. It does not matter how many times this is executed, the resulting environment will always record an aliasing between $local[\,Val]$ and $store[\,Val]$, and record the set of possible reference contexts associated with $store[\,Val]$ with the entry for $local[\,Val]$.

It is important to note that the loop summary described here is not the same as a fixed point of a loop. It may not be possible to calculate the fixed point of a loop. For instance, consider the example below, which shows a loop that instantiates new objects of a recursive data structure and does not have a static bound for its size.

```
1  Node n;
2  Node pt = new Node();
3  n = pt;
4  while(e) {
5    pt.next = new Node();
6    pt = pt.next;
7  }
```

In this example, a precise environment to capture the behaviour of the loop cannot be calculated statically. Without knowing the precise number of iterations the loop will execute, the calculated environment may not be complete. This is because the exact number of $n.next.next.next...$ fields created is not known in advance.

The environment after line 3 has been analysed in the example above is shown below (assuming that this code is executing in the mission memory area).

$$
\begin{array}{l}
\{\ \ n \mapsto n, pt \mapsto pt, n \mapsto pt, pt.next \mapsto pt.next, n.next \mapsto n.next, n.next \mapsto pt.next \\
\}\ \mapsto\ \{ \\
\quad n \mapsto \{MMem\}, pt \mapsto \{MMem\}, n.next \mapsto \{\}, pt.next \mapsto \{\} \\
\}
\end{array}
$$

At this point, the reference context in which the object referenced by n and pt is known to reside in the mission memory area. The fields $n.next$ and $pt.next$ point to null.

After analysing the while loop on lines 5 and 6 once, the resulting environment is as follows.

$$\begin{array}{l} \{ \;\; n \mapsto n, n \mapsto pt, n \mapsto pt.next, pt \mapsto pt, pt \mapsto n.next, pt.next \mapsto pt.next, \\ \quad n.next \mapsto n.next, n.next \mapsto pt.next, pt.next.next \mapsto pt.next.next, \\ \quad n.next.next \mapsto n.next.next, pt.next.next \mapsto n.next.next \\ \} \;\mapsto\; \{ \\ \quad n \mapsto \{MMem\}, pt \mapsto \{MMem\}, n.next \mapsto \{MMem\}, pt.next \mapsto \{MMem\}, \\ \quad n.next.next \mapsto \{\}, pt.next.next \mapsto \{\} \\ \} \end{array}$$

As shown in the environment above, the object referenced by $n.next$ and $pt.next$ has been instantiated in the mission memory area, and the new field $n.next.next$ is not allocated. The environment also includes the field $pt.next.next$, which should not exist. It should not exist because the pointer pt is updated to point to $pt.next$, so only pt (aliased with $n.next$) and $pt.next$ (aliased with $n.next.next$) should reside in the environment. The addition of $pt.next.next$ to the environment is a side-effect of the history element of the environment as previous aliases are not removed. The worst-case scenario for an environment that maintains history is that every expression references every other expression; in this example, the fact that $pt.next$ was once aliased with $n.next$ means that any new fields of $n.next$ will also become new fields of $pt.next$.

Whilst the history element maintains old alias relationships in the environment and subsequently may introduce false aliases, it also makes it possible to continue the analysis when a precise environment cannot be calculated without compromising the ability to detect memory-safety violations. More specifically, if every expression associated with the data structure is aliased with the others, any assignments or instantiations to members of the data structure are reflected in all other expressions. In the example above, this means that the expression $n.next$ and $pt.next$ record the set of possible reference contexts for all elements of the data structure. As will be shown later, it is not necessary to know about $n.next.next$ and $n.next.next.next$... as all the information required to detect memory-safety violations is recorded in $n.next$.

Method calls Method calls are analysed based on the method properties for each possible method that is being called. All method properties for all possible methods are applied to the environment.

$$
\begin{array}{l}
\lor\ (\exists\, mc : methodCall \\
\quad \bullet\ c = MethodCall\ mc \\
\quad \land\ CalcEnvCom(env, c, cexpr, rc, p) = CalcEnvMethod(env, mc, cexpr, rc, p))
\end{array}
$$

The result of the *CalcEnvCom* function for method calls is based on the result of the *CalcEnvMethod* function, which analyses the possible methods associated with a method call and applies the respective method properties for each to the environment.

$$
\begin{array}{|l}
CalcEnvMethod : Env \times methodCall \times LExpr \times RefCon \\
\qquad \times\ SCJmSafeProgram \rightarrow\!\!\!\!\!\!+\ Env \\
\hline
\forall\, env : Env;\ mc : methodCall;\ cexpr : LExpr;\ rc : RefCon; \\
\qquad p : SCJmSafeProgram \\
\quad \bullet\ CalcEnvMethod(env, mc, cexpr, rc, p) = \\
\qquad ApplyPossibleMethods(\\
\qquad\quad (GetMethodsFromSigs(mc.methods, p)), \\
\qquad\quad mc.args, env, cexpr, mc.le, rc, p)
\end{array}
$$

Similarly to the application of constructors to the environment in the analysis of the new instance command, the *CalcEnvMethod* function uses the *ApplyPossibleMethods* function described previously to apply the method properties of all possible methods that are associated with the method call to the environment. The *GetMethodsFromSigs* function analyses the method signatures recorded in the method call (*mc.methods*), which describes the set of possible methods that may be associated with the method call, and returns a set of corresponding methods. The method properties of each of these methods are applied to the environment in the *ApplyPossibleMethods* function.

EnterPrivateMemory The *EnterPrivateMemory* command executes the `run` method of a runnable object inside a new temporary private memory area. At translation time, a method call for the `run` method of the runnable object is defined and associated with the *EnterPrivateMemory* command.

The *EnterPrivateMemory* command is analysed in a similar way to a regular method call in the sense that the method call *mc* associated with the *EnterPrivateMemory* command is passed to the *CalcEnvMethod* function. The difference is that the method call is analysed in a lower reference context.

$$\begin{array}{l}
\lor\ (\exists\, mc : methodCall \\
\quad \bullet\ c = EnterPrivateMemory\ mc \\
\quad \land\ CalcEnvCom(env, c, cexpr, rc, p) = \\
\qquad CalcEnvMethod(env, mc, cexpr, (LowerRC\ rc), p))
\end{array}$$

The *LowerRC* function takes a reference context and returns the next lower reference context based on the original. For example, if the current reference context is $PRMem\ h1$ for some handler $h1$, the result of *LowerRC* applied to $PRMem\ h1$ is the first temporary private memory area for the handler: $TPMem(h1, 0)$.

ExecuteInAreaOf Like the *EnterPrivateMemory* command, the *ExecuteInAreaOf* command executes the `run` method of a runnable object, but in the memory area of another particular object. The method call mc associated with the *ExecuteInAreaOf* command corresponds to the `run` method of the runnable object.

Accordingly, the resulting environment after an *ExecuteInAreaOf* command has been analysed is the result of executing the method call in the reference context of a particular object, which is represented as a meta-reference context (mrc). For example, the meta-reference context $Erc\ a$ identifies that the method call mc is to be analysed in all possible reference contexts in which the object referenced by the variable a may reside.

$$\begin{array}{l}
\lor\ (\exists\, mrc : MetaRefCon;\ mc : methodCall;\ ref : ExprRefSet \\
\quad \mid ref = env.2 \\
\qquad \bullet\ c = ExecuteInAreaOf(mrc, mc) \\
\qquad \land\ CalcEnvCom(env, c, cexpr, rc, p) = \\
\qquad\quad DistEnvJoin\{rc1 : RCsFromMRC(mrc, rc, ref, cexpr) \\
\qquad\qquad \bullet\ (CalcEnvMethod(env, mc, cexpr, rc1, p))\})
\end{array}$$

The result of the *CalcEnvCom* function for *ExecuteInAreaOf* commands is defined as the distributed join of the environments resulting from analysing the method call with the *CalcEnvMethod* function in all of the possible reference contexts in which the object defined as the target execution context may reside. The *RCsFromMRC* function is used to determine the reference contexts associated with the target object. The set of all possible environments is merged to create a single environment with the *DistEnvJoin* function, which is a distributed version of the *EnvJoin* function.

ExecuteInOuterArea The *ExecuteInOuterArea* command, which executes the `run` method of a runnable object in the immediately outer memory area, is analysed in a similar way to the *EnterPrivateMemory* command. The difference is that the resulting environment is the result of analysing the method call in the next outer reference context, as opposed to the next lower reference context.

GetMemoryArea In SCJ, the `getMemoryArea` method returns a reference to the memory area object in which the expression passed as an argument resides. In *SCJ-mSafe*, it is the set of reference contexts in which the object referenced by the expression passed as an argument that are returned. This is subtly different as the behaviour in *SCJ-mSafe* returns the associated reference context that describes the memory area as opposed to the memory area object itself. The resulting environment after analysing a *GetMemoryArea* command is defined by the *CalcEnvGetMemArea* function.

$$\begin{array}{l}
\lor\ (\exists\, gma : getMemoryArea \\
\quad \bullet\ c = GetMemoryArea\ gma \\
\quad \land\ CalcEnvCom(env, c, cexpr, rc, p) = \\
\qquad CalcEnvGetMemArea(env, gma, cexpr, rc, p))
\end{array}$$

The *CalcEnvGetMemArea* function defines the environment so that the reference component *gma.ref* of the *GetMemoryArea* command points to the possible reference contexts in which the object referenced by the expression *gma.e* resides. The function takes the current environment, a *getMemoryArea* command, a left expression that defines the current execution point, the current reference context, and the *SCJ-mSafe* program being analysed as its parameters, and returns an updated environment.

$$\begin{array}{l}
CalcEnvGetMemArea : Env \times getMemoryArea \times LExpr \\
\qquad \times\ RefCon \times SCJmSafeProgram \rightarrow\!\!\!\!\!+ Env \\
\hline
\forall\, env : Env;\ gma : getMemoryArea;\ cexpr : LExpr; \\
\qquad rc : RefCon;\ p : SCJmSafeProgram \\
\quad \bullet\ \exists\, rel : ExprShareRelation;\ ref : ExprRefSet \mid env = (rel, ref) \\
\qquad \bullet\ \exists\, newRef, newExpr : LExpr;\ erc : MetaRefCon \\
\qquad\quad \mid newRef = MergeExprs(cexpr, gma.ref, p) \\
\qquad\quad \land\ (gma.e = This \land newExpr = cexpr \\
\qquad\qquad \lor\ gma.e \neq This \land newExpr = MergeExprs(cexpr, gma.e, p)) \\
\qquad\quad \land\ erc = Erc\ newExpr \\
\qquad\qquad \bullet\ CalcEnvGetMemArea(env, gma, cexpr, rc, p) = \\
\qquad\qquad rel \mapsto ExprRefUpdate(newRef, \\
\qquad\qquad\quad (RCsFromMRC(erc, rc, ref, cexpr)), ref)
\end{array}$$

The *CalcEnvGetMemArea* function first defines the updated reference *ref* and expression *e* based on the current expressions with the *MergeExprs* function (lines 7-9). The updated expressions are recorded in the variables *newRef* and *newExpr* (defined on line 6), respectively.

The overall result of the function is a mapping between the unchanged expression share relation *rel* and the updated expression reference set, which is updated with the *ExprRefUpdate* function (lines 11-13). The expression reference set is updated with a mapping from the new reference *newRef* to the set of possible reference contexts associated with the meta-reference context *Erc newExpr*, which is obtained with the *RCsFromMRC* function.

Try The *Try* command is analysed based on the try statement, all possible catch statements, and the finally statement. The resulting environment is a summary of all possible behaviours.

```
1  $\lor (\exists c1, c2 : Com;\ eseq : \mathrm{seq}\, Expr;\ comseq : \mathrm{seq}\, Com$
2      $\bullet\ c = Try(c1, eseq, comseq, c2)$
3      $\land\ CalcEnvCom(env, c, cexpr, rc, p) =$
4          $EnvJoin($
5              $(EnvJoin($
6                  $(CalcEnvCom(env, c1, cexpr, rc, p)),$
7                  $(DistEnvJoin\{com : \mathrm{ran}\, comseq$
8                      $\bullet\ (CalcEnvCom(env, com, cexpr, rc, p))\}))),$
9              $(CalcEnvCom(env, c2, cexpr, rc, p))))$
```

The result of the *CalcEnvCom* function for the *Try* command is the merged result of analysing the try component of the *Try* command ($c1$) on line 6, the distributed join of all possible catch commands in *comseq* on lines 7 and 8, and the finally component $c2$ on line 9.

4.4.2 Handlers

The resulting environment after a handler has executed is based on the execution of the `handleEvent` method. The *CalcEnvHandler* function below describes how the environment is updated for an individual handler.

$$CalcEnvHandler : Env \times Handler \times LExpr \times SCJmSafeProgram \rightarrowtail Env$$

$$\begin{array}{l}
\forall env : Env;\ h : Handler;\ cexpr : LExpr;\ p : SCJmSafeProgram \\
\quad \bullet\ CalcEnvHandler(env, h, cexpr, p) = \\
\quad\quad RemoveExprSetEnv((LocalVars\ h.hAe), \\
\quad\quad (CalcEnvCom(env, h.hAe, cexpr, (PRMem\ h.name), p)))
\end{array}$$

The *CalcEnvHandler* function takes an environment *env*, a handler *h*, the current expression *cexpr*, and the *SCJ-mSafe* program *p* being analysed, and returns an updated environment. The result of the function is based on the result of *CalcEnvCom*, which analyses the command *h.hAe*, which is the `handleEvent` method of the handler. The fourth parameter passed to the *CalcEnvCom* function is the reference context in which the command is to be analysed; in the case of the `handleEvent` method, this is the per-release memory area associated with the handler (*PRMem h.name*).

The environment resulting from the *CalcEnvCom* function call is then restricted with the *RemoveExprSetEnv* function, which removes all of the local expressions defined in a particular *SCJ-mSafe* construct, from the environment. In this case, the local variables of the *h.hAe* component are removed. The local variables are calculated with the *LocalVars* function, which takes a command and returns a set of expressions that describe the local variables declared as part of the command.

The resulting environment is used as the environment to analyse subsequent handlers; it does not contain any expressions local to the handler that has been analysed, but it does contain information about the changes to any shared objects between handlers.

To calculate the resulting environment for all of the handlers associated with a particular mission, the *CalcEnvHandlers* function is defined.

$$CalcEnvHandlers : Env \times Mission \times \mathbb{P}\ Handler \times CheckingEnv \rightarrowtail Env$$

$$\begin{array}{l}
\forall env : Env;\ m : Mission;\ handlers : \mathbb{P}\ Handler;\ p : SCJmSafeProgram \\
\quad \bullet\ CalcEnvHandlers(env, m, handlers, p) = \\
\quad\quad DistEnvJoin\{h : handlers \\
\quad\quad\quad \bullet\ (CalcEnvHandler(env, h, GetHandlerExpr(p, h, m), p))\}
\end{array}$$

This function takes an old environment *env*, a mission *m*, the set of handlers associated with the mission *handlers*, and the *SCJ-mSafe* program *p* being analysed, and returns an updated environment. The result of the function is the merged result of analysing all handlers in the set of handlers with the *CalcEnvHandler* function; this is done with the *DistEnvJoin* function. The third parameter of the *CalcEnvHandler* function is the current expression, which in the case of a specific handler is the left expression that references the handler object. This is calculated with the *GetHandlerExpr* function, which is a syntactic function on the *SCJ-mSafe* program; it takes the *SCJ-mSafe* program, the handler, and the mission as its parameters, and returns an expression that references the handler object.

The resulting environment obtained from the *CalcEnvHandlers* function is a summary of the behaviour of all handlers associated with a particular mission. This is then used by the *CalcEnvMission* function that will be described next.

4.4.3 Missions

The resulting environment after a mission has executed is based on the execution of the `initialize` method, the associated handlers, and the `cleanUp` method. The *CalcEnvMission* function below describes how the environment is updated for an individual mission.

```
1  $CalcEnvMission : Env \times Mission \times LExpr \times SCJmSafeProgram \rightarrow\!\!\!\!\!+ Env$
   ────────────
2  $\forall env : Env;\ m : Mission;\ cexpr : LExpr;\ p : SCJmSafeProgram$
3    $\bullet$ let $initializeEnv == CalcEnvCom(env, m.initialize, cexpr, MMem, p)$
4      $\bullet$ let $handlersEnv == CalcEnvHandlers($
5          $(RemoveExprSetEnv($
6            $(LocalVars\ m.initialize \setminus GetHandlerExprs(p, m)),$
7                $initializeEnv)),$
8            $m, (GetHandlers(p, m.handlers)), p)$
9        $\bullet$ let $cleanUpEnv == CalcEnvCom($
10         $(RemoveExprSetEnv((LocalVars\ m.initialize), handlersEnv)),$
11         $m.cleanUp, cexpr, MMem, p)$
12         $\bullet\ CalcEnvMission(env, m, cexpr, p) =$
13           $RemoveExprSetEnv((LocalVars\ m.cleanUp), cleanUpEnv)$
```

The *CalcEnvMission* function takes an old environment env, a mission m, the current expression $cexpr$, and the *SCJ-mSafe* program p being analysed, and returns an updated environment.

The function specification declares three local variables: *initializeEnv* (line 3), *handlersEnv* (line 4), and *cleanUpEnv* (line 9), which correspond to the resulting environments after the `initialize` method has been analysed, the handlers associated with the mission have been analysed, and the `cleanUp` method has been analysed.

The *initializeEnv* is the result of the *CalcEnvCom* function applied to the $m.initialize$ command in the current environment env. The *handlersEnv* is the result of the

CalcEnvHandlers function, when applied to *initializeEnv* as its environment, but removes the local variables from the *m.initialize* command (as they have gone out of scope) except for the expressions that identify the handlers. Finally, the *cleanUpEnv* is the result of the *CalcEnvCom* function, which uses the *handlersEnv* as its environment, but removes the remaining local variables from the *m.initialize* command, which includes all of the expressions that identify handler objects.

The overall result of the function is the *cleanUpEnv* with the local variables from the *m.cleanUp* command removed. The resulting environment is used as the environment to analyse subsequent missions; it does not contain any expressions local to the mission that has been analysed, but it does contain information about the changes to any shared objects between missions.

To calculate the resulting environment after a set of missions have executed, the *CalcEnvMissions* function is defined.

$$
\begin{array}{l}
CalcEnvMissions : Env \times \mathbb{P}\, Mission \times SCJmSafeProgram \rightarrow\!\!\!\!\!\!+\;\; Env \\
\hline
\forall\, env : Env;\; missions : \mathbb{P}\, Mission;\; p : SCJmSafeProgram \\
\quad \bullet\; CalcEnvMissions(env, missions, p) = \\
\qquad DistEnvJoin\{m : missions \\
\qquad\quad \bullet\; (CalcEnvMission(env, m, ((GetMissionExpr(p, m), p))\}
\end{array}
$$

The *CalcEnvMissions* function takes an old environment *env*, a set of missions *missions*, and the *SCJ-mSafe* program *p* being analysed, and returns an updated environment. The result of the function is the distributed join of all missions analysed with the *CalcEnvMission* function. The third parameter of *CalcEnvMission* is the current expression, which in the case of a specific mission is the left expression that references the mission object. Like the handler objects, this is also calculated with a syntactic function (*GetMissionExpr*) on the *SCJ-mSafe* program.

4.4.4 Mission Sequencers and Safelets

There is no function to calculate the resulting environment after a mission sequencer or safelet has executed. This is because when analysing a program, there is no further execution after a mission sequencer has finished executing; therefore, the resulting environment would not be used. Similarly, the safelet is the top-level component in a program, and nothing would be able to make use of the resulting environment.

All of these *CalcEnv* functions are used during the checking phase of the technique, which is described in more detail in Section 4.6.

4.5 Generating method properties

The method properties for each method in an *SCJ-mSafe* program are generated after the translation phase, and before the checking phase of the technique. The generation of method properties is similar to the updating of the environment presented above. For example, there exists a *CalcPropertiesCom* function that defines what happens to method properties when a particular command is analysed. The main difference is the use of meta-reference contexts as opposed to reference contexts; the interesting differences in the calculation are presented here.

4.5.1 Commands

The *CalcPropertiesCom* function specifies the behaviour of a command and returns an updated method properties and set of local variables based on the changes made. The local variables for each method are calculated for the checking phase of the technique. As method properties are only given a context when they are added to an environment, all local variables that a method call introduces must be maintained in the method properties, as they cannot be checked until the context of the method call is known. When they have been checked against the calling context, they can be removed from the environment as they are no longer in scope; maintaining the set of local variables during the generation of method properties is therefore necessary to identify the variables that go out of scope. The set of local variables also takes into account the local variables of additional methods that are called from the current method being analysed. This is because, as mentioned above, the calling context is not known at this point.

$$
\begin{array}{l}
\mathit{CalcPropertiesCom} : \mathit{Method} \times \mathit{MethodProperties} \times \mathbb{P}\,\mathit{LExpr} \\
\quad \times\ \mathit{Com} \times \mathit{LExpr} \times \mathit{MetaRefCon} \times \mathit{SCJmSafeProgram} \\
\qquad \pfun \mathit{MethodProperties} \times \mathbb{P}\,\mathit{LExpr} \\
\hline
\forall\, m : \mathit{Method};\ \mathit{properties} : \mathit{MethodProperties};\ \mathit{localVars} : \mathbb{P}\,\mathit{LExpr}; \\
\quad c : \mathit{Com};\ \mathit{cexpr} : \mathit{LExpr};\ \mathit{mrc} : \mathit{MetaRefCon};\ p : \mathit{SCJmSafeProgram} \\
\qquad \bullet \ldots
\end{array}
$$

The function takes as its arguments the method that is being analysed (m), the old method properties (*properties*), the old set of local variables (*localVars*), the command to be analysed (c), the current expression ($cexpr$), the current meta-reference context (mrc), and the *SCJ-mSafe* program being analysed (p), and returns an updated method properties and set of local variables.

Declarations Declarations are analysed just like in the *CalcEnvCom* function; an additional function *AddDecToMethodProperties* is used to add the new mapping to the method properties. If the new declaration is of a variable of a primitive type, the set of meta-reference contexts is $\{Rcs\{Prim\}\}$, which identifies that the precise reference context of the expression is known, and is the primitive context in this case. If the declaration is not of primitive type, the set of meta-reference contexts is empty, because the variable is uninitialised at this point.

$$
\begin{array}{l}
\lor (\exists d : Dec \\
\quad \bullet c = Decl\ d \\
\quad \land CalcPropertiesCom(m, properties, localVars, c, cexpr, mrc, p) = \\
\quad\quad (AddDecToMethodProperties(properties, d), \\
\quad\quad\quad localVars \cup \{MergeExprs(cexpr, (ID(var\ d.var)), p)\}))
\end{array}
$$

The additional part of the declaration command analysis in *CalcPropertiesCom* is the definition of *localVars*, which includes the declaration expression after it has been updated with the current expression in the *MergeExprs* function. This is how the set of local variables in a method is recorded. As mentioned above, this is necessary to identify the expressions that, once checked, go out of scope in the method properties when a method is called.

Assignments Assignments in *CalcPropertiesCom* are analysed in exactly the same way as *CalcEnvCom*, that is, the expression share relation and method reference set are updated based on the mapping from the left expression to the right expression, and all subsequent fields and equal expression in the method properties are updated. The one difference is how the method reference set is updated. When updating the environment, the set of reference contexts associated with the left expression is defined as the set of reference contexts associated with the right expression in the environment. This is because the left expression now points to the same object as the right expression, and it resides in the possible reference contexts already recorded.

When calculating the changes to method reference sets, it may not be possible to determine the set of meta-reference contexts of the right-hand side of the expression, because the right-hand expression may not be in the method properties. This is because the right expression may be a global variable or field of the target object, for example. In the case where the set of possible meta-reference contexts cannot be determined, because there is no entry in the method properties, the *MethodRefAddPropertiesAsgn* function uses the *Erc* meta-reference context to define that the referenced object resides in the expression reference context of another expression that is not currently in scope.

$$
\begin{array}{l}
\mathit{MethodRefAddPropertiesAsgn} : \mathit{LExpr} \times \mathit{Expr} \times \mathit{MethodProperties} \\
\qquad \pfun \mathit{MethodProperties} \\
\hline
\forall \mathit{properties} : \mathit{MethodProperties};\ \mathit{le} : \mathit{LExpr};\ \mathit{re} : \mathit{Expr} \\
\quad \bullet \exists \mathit{rel} : \mathit{ExprShareRelation};\ \mathit{ref} : \mathit{MethodRefSet} \mid \mathit{properties} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet \mathit{re} \in \operatorname{dom} \mathit{ref} \\
\qquad\quad \land \mathit{MethodRefAddPropertiesAsgn}(\mathit{le}, \mathit{re}, \mathit{properties}) = \\
\qquad\qquad \mathit{rel} \mapsto \mathit{MethodRefUpdate}(\mathit{le}, (\mathit{ref}\ \mathit{re}), \mathit{ref}) \\
\qquad \lor \mathit{re} \notin \operatorname{dom} \mathit{ref} \\
\qquad\quad \land \mathit{MethodRefAddPropertiesAsgn}(\mathit{le}, \mathit{re}, \mathit{properties}) = \\
\qquad\qquad \mathit{rel} \mapsto \mathit{MethodRefUpdate}(\mathit{le}, \{(\mathit{Erc}\ \mathit{re})\}, \mathit{ref})
\end{array}
$$

There are two cases for the result of *MethodRefAddPropertiesAsgn*. The first is when the right expression *re* is in the domain of the method reference set *ref* ($re \in \operatorname{dom} ref$), and the second is when it is not in the domain ($re \notin \operatorname{dom} ref$). If the right expression is in the domain of the method reference set, the result is the set of reference contexts associated with the right expression ($ref\ re$), which is defined in the method reference set by the *MethodRefUpdate* function.

If not, the method properties calculated so far does not have information on the location of the right expression; this occurs when the right expression is defined outside of the scope of the method body. The result, therefore, is the meta-reference context *Erc* that states the set of reference contexts of the left expression is defined by those of the right expression *re*, which is unknown at this point. *Erc* meta-reference contexts are resolved when the properties are added to an environment, as the right expression of the assignment will be in scope at this point.

NewInstance Like in the *CalcEnvCom* function, the new instance command has its own specific function when analysing method properties. The *CalcPropertiesNewInstance* function is very similar to the *CalcEnvNewInstance* function; the main difference is that the new expression is instantiated in a meta-reference context as opposed to a reference context. The update of method properties is the same as the environment: the method reference set is updated with a mapping from the left expression to the set of possible meta-reference contexts in which the object is being instantiated, and the method properties of the relevant constructor are applied.

In the *CalcEnvNewInstance* function, the *RCsFromMRC* function determines which reference contexts the object may reside in based on the meta-reference context of the new instance command ($nI.mrc$). In this case, the meta-reference context of the new instance command is used to determine the set of meta-reference contexts in which the object may reside (as opposed to reference contexts), and is calculated with the *AnalyseMetaRefCon* function.

$$
\begin{array}{|l}
AnalyseMetaRefCon : MetaRefCon \times MetaRefCon \times MethodRefSet \\
\qquad \pfun \mathbb{P}\, MetaRefCon \\
\hline
\forall\, nImrc, mrc : MetaRefCon;\ ref : MethodRefSet \\
\quad \bullet (\exists\, n : \mathbb{N} \\
\qquad nImrc = Current \\
\qquad\quad \land AnalyseMetaRefCon(nImrc, mrc, ref) = \{mrc\} \\
\qquad \lor (\exists\, n : \mathbb{N} \\
\qquad\quad \mid nImrc = CurrentPrivate\ n \\
\qquad\qquad \bullet ((\exists\, n1 : \mathbb{N} \\
\qquad\qquad\quad \mid mrc = CurrentPlus\ n1 \\
\qquad\qquad\qquad \bullet\ n1 - n > 0 \land AnalyseMetaRefCon(nImrc, mrc, ref) = \\
\qquad\qquad\qquad\quad \{CurrentPlus(n1 - n)\} \\
\qquad\qquad\qquad \lor n1 - n < 0 \land AnalyseMetaRefCon(nImrc, mrc, ref) = \\
\qquad\qquad\qquad\quad \{CurrentPrivate(n - n1)\} \\
\qquad\qquad\qquad \lor n1 - n = 0 \land AnalyseMetaRefCon(nImrc, mrc, ref) = \\
\qquad\qquad\qquad\quad \{Current\}) \\
\qquad\qquad\quad \lor mrc = Current \land AnalyseMetaRefCon(nImrc, mrc, ref) = \\
\qquad\qquad\qquad\quad \{CurrentPrivate\ n\} \\
\qquad\qquad\quad \lor (\exists\, n1 : \mathbb{N} \\
\qquad\qquad\qquad \mid mrc = CurrentPrivate\ n1 \\
\qquad\qquad\qquad\quad \bullet AnalyseMetaRefCon(nImrc, mrc, ref) = \\
\qquad\qquad\qquad\qquad \{CurrentPrivate(n1 + n)\}))) \\
\qquad \lor \ldots
\end{array}
$$

The *AnalyseMetaRefCon* function takes two meta-reference contexts and a method reference set, and returns a set of possible meta-reference contexts. The first meta-reference context ($nImrc$) is the meta-reference context associated with the new instantiation, and the second (mrc) is the current meta-reference context. The extract from the function above shows that if the new instance meta-reference context is equal to *Current*, then the new object resides in the set of meta-reference contexts that contains only the current meta-reference context (mrc).

For example, when the analysis of a method body begins, the initial current meta-reference context is *Current*. If a new instance command then instantiates a new object in the *Current* meta-reference context, the result is that the new object resides in the *Current* reference context. If, however, during the method body a temporary private memory area is entered, the current meta-reference context would be $\mathit{CurrentPrivate}(0)$. If a new instance command instantiates a new object in the *Current* reference context now, the new object will reside in the $\mathit{CurrentPrivate}(0)$ reference context, as that is the current context at the specific point of the analysis.

If, however, the new instance meta-reference context is equal to $\mathit{CurrentPrivate}(n)$, then the result of the function is a meta-reference context that is lower then the current meta-reference context by depth n. For example, if the current meta-reference context is *Current* and the new instance meta-reference context is $\mathit{CurrentPrivate}(n)$, the resulting meta-reference context is simply $\mathit{CurrentPrivate}(n)$. If the current meta-reference context is also a *CurrentPrivate* context with depth m, the resulting meta-reference context is *CurrentPrivate* with depth $(m + n)$.

If the current context is $\mathit{CurrentPlus}(m)$, then the resulting meta-reference context is calculated by subtracting the depth of the new instance meta-reference context $\mathit{CurrentPrivate}(n)$ from the depth of the current meta-reference context $\mathit{CurrentPlus}(m)$. If $(m - n)$ is greater than zero, the result is $\mathit{CurrentPlus}(m - n)$, if it is less than zero the result is $\mathit{CurrentPrivate}(n - m)$, and if it is exactly zero then the result is *Current*.

4.5.2 Building all method properties

After an SCJ program has been translated, the resulting *SCJ-mSafe* program contains all of the classes and methods of the original SCJ program. These methods are translations of the SCJ methods, and have not been analysed. The *BuildMethodProperties* function defined below analyses all methods inside an *SCJ-mSafe* program and returns an updated *SCJ-mSafe* program whose methods now include the calculated method properties, local variables, and visible fields, which are fields of the containing class (and any classes it inherits from) that are in scope during the method's execution. The input *SCJ-mSafe* program, which does not contain the generated method properties, is rebuilt by the *BuildMethodProperties* function by a distributed application of the function *BuildMethodPropertiesMethod* that specifies the method properties for a single method.

The output of the *BuildMethodProperties* function is essentially an *SCJ-mSafe* program that has been automatically annotated with the method properties for each method in the translated program.

```
1  $BuildMethodProperties : SCJmSafeProgram \pfun SCJmSafeProgram$
   ----------------
2  $\forall p : SCJmSafeProgram$
3      $\bullet \textbf{let } methods == p.safelet.methods \cup p.missionSeq.methods$
4          $\cup \bigcup\{m : p.missions \bullet m.methods\}$
5          $\cup \bigcup\{h : p.handlers \bullet h.methods\}$
6          $\cup \bigcup\{c : p.classes \bullet c.methods\}$
7          $\bullet \exists methodSeq : \mathrm{seq}\, Method;\ analysedMethods : \mathbb{P}\, Method;$
8              $deps : MethodDependencies;\ p' : SCJmSafeProgram$
9              $\mid \mathrm{ran}\, methodSeq = methods \land \#\, methodSeq = \#\, methods$
10             $\land analysedMethods = \mathrm{ran}(BuildMethodPropertiesMethods($
11                 $(SortMethods(methodSeq, deps)), p))$
12             $\land deps = GetDeps\ p$
13                 $\bullet p'.static = p.static$
14                 $\land p'.sInit = p.sInit$
15                 $\land p'.safelet =$
16                     $BuildMethodPropertiesSafelet(p.safelet, analysedMethods)$
17                 $\land p'.missionSeq =$
18                     $BuildMethodPropertiesMSeq(p.missionSeq, analysedMethods)$
19                 $\land p'.missions =$
20                     $BuildMethodPropertiesMissions(p.missions, analysedMethods)$
21                 $\land p'.handlers =$
22                     $BuildMethodPropertiesHandlers(p.handlers, analysedMethods)$
23                 $\land p'.classes =$
24                     $BuildMethodPropertiesClasses(p.classes, analysedMethods)$
25                 $\land BuildMethodProperties\ p = p'$
```

The function takes an *SCJ-mSafe* program (p) and returns an updated *SCJ-mSafe* program. The set of all methods in an *SCJ-mSafe* program (*methods*) is defined as the union of the methods from all classes (lines 3-6). The function definition then states that there must exist a sequence of methods (*methodSeq*) such that every method in the set of methods *methods* is in the sequence (line 9); this sequence defines the basis for the ordering in which the methods are analysed.

The order in which the methods are analysed is based on the method dependencies (*deps*), which is a relation that describes the call graph for methods. The dependencies relation *deps* is calculated with the *GetDeps* function, which takes an *SCJ-mSafe* program and returns a relation (line 12).

There must also exist another set of methods (*analysedMethods*), which records the methods after their method properties have been calculated. The set of analysed methods is calculated with the *BuildMethodPropertiesMethods* function.

The *BuildMethodPropertiesMethods* function is used to generate the method properties for a sequence of methods; the first parameter passed to the function is the result of the *SortMethods* function, which analyses the method dependencies (*deps*) and returns a sequence in which the methods can be successfully analysed. Methods that have no dependencies on other methods, that is they do not call other methods as part of their execution, are analysed first. Next, methods whose dependents have been analysed are added to the sequence; this continues until all methods have a place in the analysis sequence.

If a method is dependent on itself, or more specifically, it is a recursive method, it is analysed in the sequence like all other methods once all other dependents have been analysed. Methods that are dependent on each other, that is, are mutually recursive, are not handled in the technique; a discussion as to how they may be incorporated is included in Section 6.3.

The resulting *SCJ-mSafe* program p' returned by *BuildMethodProperties* remains the same as the input program p, except for the additional method properties that have been added to each method. The *BuildMethodProperties* functions on lines 16-24 are used to update specific components with the corresponding methods for that component in *analysedMethods*.

Individual methods are defined using the *BuildMethodPropertiesMethod* function, which takes a method and the *SCJ-mSafe* program being analysed and returns an updated method.

$$
\begin{array}{l}
\mathit{BuildMethodPropertiesMethod} : \mathit{Method} \times \mathit{SCJmSafeProgram} \pfun \mathit{Method} \\
\hline
\forall \mathit{method} : \mathit{Method};\ p : \mathit{SCJmSafeProgram} \\
\quad \bullet \exists \mathit{method}' : \mathit{Method} \\
\qquad \mid \mathit{method}'.\mathit{name} = \mathit{method}.\mathit{name} \\
\qquad \land \mathit{method}'.\mathit{returnType} = \mathit{method}.\mathit{returnType} \\
\qquad \land \mathit{method}'.\mathit{type} = \mathit{method}.\mathit{type} \\
\qquad \land \mathit{method}'.\mathit{params} = \mathit{method}.\mathit{params} \\
\qquad \land \mathit{method}'.\mathit{class} = \mathit{method}.\mathit{class} \\
\qquad \land \mathit{method}'.\mathit{body} = \mathit{method}.\mathit{body} \\
\qquad \land \mathit{method}'.\mathit{properties} = (\mathit{CalcPropertiesCom}(\mathit{method}, (\varnothing, \varnothing), \\
\qquad\quad \varnothing, \mathit{method}.\mathit{body}, \mathit{Null}, \mathit{Current}, p)).1 \\
\qquad \land \mathit{method}'.\mathit{localVars} = (\mathit{CalcPropertiesCom}(\mathit{method}, (\varnothing, \varnothing), \\
\qquad\quad \varnothing, \mathit{method}.\mathit{body}, \mathit{Null}, \mathit{Current}, p)).2 \\
\qquad \land \mathit{method}'.\mathit{visibleFields} = \mathit{AnalyseMethodVisibleFields}(\mathit{method}, p) \\
\qquad\quad \bullet \mathit{BuildMethodPropertiesMethod}(\mathit{method}, p) = \mathit{method}'
\end{array}
$$

The input *method* and *SCJ-mSafe* program p are used in the calculation of an updated method (method'), which is the overall result of the function. The updated method remains the same as the input method, except for the method properties, local variables, and visible fields, which are calculated by the function. The method properties and local variables are defined using the *CalcPropertiesCom* function described previously. The arguments given are the method being analysed (*method*), an empty method properties $(\varnothing, \varnothing)$, an empty set of local variables $(\varnothing)$, the command of the method body ($\mathit{method}.\mathit{body}$), the current expression, which is *Null* as no execution has been analysed yet, the meta-reference context *Current* (as the initial default allocation context is the current context), and the *SCJ-mSafe* program.

The *CalcPropertiesCom* function returns a pair, as defined previously. The first element of the pair is the method properties, which is recorded in $\mathit{method}'.\mathit{properties}$, and the second element is the set of local variables that have been defined as part of the method's execution, which are recorded in $\mathit{method}'.\mathit{localVars}$.

Finally, the visible fields of the method are calculated using the *AnalyseMethodVisibleFields* function, which takes the current method being analysed and the *SCJ-mSafe* program, and returns a set of left expressions that defines the fields of the target object that are in scope when the method is executed.

The generation of method properties described in this section and the functions defined to update the environment provide the necessary tools to check *SCJ-mSafe* programs for possible memory-safety violations. The remainder of this chapter describes how the checking of programs is performed.

4.6 Rules for checking *SCJ-mSafe* programs

This section defines the memory-safety rules of *SCJ-mSafe* programs; rules exist for all components of *SCJ-mSafe* from the top-level overall program to commands. There is also a rule that defines what it means for an environment to be memory safe, which is based on the checking technique described above.

Each rule has a set of hypotheses and a conclusion. All hypotheses must be true in order for the conclusion to be true. If one or more hypothesis of a rule is false, then there exists possible memory-safety violations in the component associated with the rule.

The rules presented in this section are designed to give a readable description of the underlying formalisation that can be found in Appendix E. The presentation of the first rule includes a description of the associated formalisation; a discussion of the remaining formalisations is omitted.

This section first defines the *Dominates* relation, which is an ordering on the reference contexts of an *SCJ-mSafe* program. The memory safety inference rule for environments is then presented before the rules for all *SCJ-mSafe* commands are defined. The rule for an overall *SCJ-mSafe* program is then presented before the individual rules for each *SCJ-mSafe* component.

4.6.1 The dominates relation

In order to establish whether a reference from one reference context to another is safe, an ordering on the reference contexts used in the environment is defined; this is called the *Dominates* relation. It defines which reference contexts dominate others, or more specifically, which reference contexts are higher in the memory-structure hierarchy.

$$
\begin{array}{|l}
\mathit{Dominates} : \mathit{RefCon} \leftrightarrow \mathit{RefCon} \\
\hline
\mathit{Dominates} = \{(\mathit{Prim} \mapsto \mathit{IMem}), \\
\quad (\mathit{IMem} \mapsto \mathit{MMem}), \\
\quad (\mathit{MMem} \mapsto \mathit{TPMMem}\,0)\} \\
\quad \cup \{h : \mathit{LExpr} \bullet (\mathit{MMem} \mapsto \mathit{PRMem}\,h)\} \\
\quad \cup \{h : \mathit{LExpr} \bullet (\mathit{PRMem}\,h \mapsto \mathit{TPMem}(h, 0))\} \\
\quad \cup \{h : \mathit{LExpr};\ x : \mathbb{N} \bullet (\mathit{TPMem}(h, x) \mapsto \mathit{TPMem}(h, (x + 1)))\} \\
\quad \cup \{x : \mathbb{N} \bullet (\mathit{TPMMem}\,x \mapsto \mathit{TPMMem}(x + 1))\}
\end{array}
$$

The mappings in the relation define a hierarchy of *SCJ-mSafe* reference contexts. At the top of the hierarchy is the primitive reference context Prim, which dominates the immortal memory IMem. Next, the immortal memory area IMem dominates the mission memory area MMem.

The mission memory MMem dominates the first temporary private mission memory area $\mathit{TPMMem}\,0$, and also all the per-release memory areas for the handlers $\mathit{PRMem}\,h$. Like the mission memory area, the per-release memory area for each handler dominates the first temporary private memory area associated with the corresponding handler ($\mathit{PRMem}\,h \mapsto \mathit{TPMem}(h, 0)$).

Finally, all temporary private mission memory areas and temporary private memory areas of a handler dominate further nested temporary private mission memory areas and temporary private memory areas respectively. For example, $\mathit{TPMem}(h, 2)$ dominates $\mathit{TPMem}(h, 3)$ as the third temporary private memory area is more nested than the second.

The $\mathit{Dominates}$ relation defines the hierarchy of the *SCJ-mSafe* reference contexts, however, the reflexive transitive closure of the $\mathit{Dominates}$ relation gives a complete definition of which reference contexts dominate others. For example, there is no mapping from the immortal memory area IMem to a temporary private memory area $\mathit{TPMem}(h, 1)$, however, this mapping is included in through the transitive closure. The reflexive closure is also taken as reference contexts dominate themselves; this is because a reference from one object to another in the same memory area is safe.

Using the $\mathit{Dominates}$ relation above, it is possible to check the environment throughout the analysis to detect possible memory-safety violations. A memory-safety violation may occur if a reference variable or a field of an object, points to an object that resides in a lower reference context than the one in which it is defined. Checking for possible violations, therefore, requires a traversal of the environment for each element to determine whether any fields of that element reside in lower reference contexts.

4.6.2 Environment

The first inference rule presented is the rule that defines what it means for an environment to be memory safe (*mSafeEnv*). This is the lowest-level rule and is the point at which possible memory-safety violations are detected.

The hypotheses of the *mSafeEnv* rule state that the static variables, object fields, and local variables must all be safe for the overall environment to be memory safe. In summary, objects referenced by static variables must reside in the immortal memory or be of a primitive type. Object fields must reside in a reference context that is equal to or higher than that of the containing object. Finally, the local variables of the component currently being analysed must reside in a reference context that is equal to or higher than the reference context in which the variable was declared. The set of local variables is a parameter of the rule as it is updated throughout the checking procedure, as will be demonstrated in the rules for commands in the next section.

Consider, for example, the extract of code for a handler shown below.

```java
public MyHandler extends PeriodicEventHandler {
    static Object staticVar;
    Object fieldVar;

    public void handleAsyncEvent() {
        Object localVar;
        ...
```

In the example above, the static variable `staticVar` is checked against the immortal memory area, as the referenced object must reside in that memory area and no lower. The field `fieldVar` is checked against the reference context of the containing object, which in this case is the instance of the handler. The handler object resides in the mission memory area, therefore the object referenced by the field must reside in either the mission or immortal memory area. Finally, the local variable `localVar` is checked against the reference context in which it was defined, which during the analysis of the `handleAsyncEvent` method is the per-release memory area associated with the handler. Therefore the object referenced by the local variables must reside in either the per-release memory area of the handler, the mission memory area, or the immortal memory area. The *mSafeEnv* rule is shown below.

$$\dfrac{\begin{array}{l}
\forall (le_1, refSet_1) : ref \bullet \\
\quad le_1 \in GetStaticVars(p) \\
\quad \land\ DominatesLeast(refSet_1) \mapsto IMem \in Dominates^* \\
\land\ \forall (le_1, refSet_1), (le_2, refSet_2) : ref \mid \\
\quad FieldOf(le_1, le_2) \bullet \\
\quad DominatesLeast(refSet_1) \mapsto DominatesTop(refSet_2) \in Dominates^* \\
\land\ \forall (le_1, refSet_1) : ref \bullet \\
\quad le_1 \in localVars \\
\quad \land\ DominatesLeast(refSet_1) \mapsto rc \in Dominates^* \\
\land\ \forall (le_1, refSet_1) : ref \mid \\
\quad le_1 \notin GetStaticVars(p) \land le_1 \notin localVars \\
\quad \land\ (\neg\ \exists (le_2, refSet_2) : ref \bullet FieldOf(le_1, le_2)) \bullet \\
\qquad \exists (le_3, refSet_3) \mid le_3 = longestPrefixOf(env, le_1) \bullet \\
\qquad\quad \land\ DominatesLeast(refSet_1) \mapsto DominatesTop(refSet_3) \in Dominates^*
\end{array}}{mSafeEnv(env, localVars, rc, p)}$$

where
$env = (share, ref)$

The conclusion of the memory-safety rule states that a particular environment *env* is memory safe with the current local variables *localVars*, in the current reference context *rc*, and *SCJ-mSafe* program *p* if the hypotheses are true. The static variables in the environment must be safe to satisfy the first hypothesis. All fields of objects must be safe to satisfy the second, and all local variables must be safe to satisfy the third. Expressions that have not been checked by the first three hypotheses, are checked with the fourth and final hypothesis, which is used to check incomplete environments, as will be explained in what follows.

Static variables Static variables cannot be compared against the reference context of a containing object as there is none. When checking each expression in the environment, if the expression is a static reference variable, the reference context of the object referenced must be the immortal memory area, otherwise a memory-safety violation may occur.

The first hypothesis states that for every expression in the reference set (le_1) that is a static reference variable, the worst-case mapping from the lowest possible reference context of the expression to the immortal memory area is in the reflexive transitive closure of the *Dominates* relation. More specifically, this states that all objects referenced by static variables must reside in the immortal reference context.

As the precise reference context in which an object resides cannot always be determined, the worst-case mapping from one reference context to another is analysed. The lowest possible reference context of the object referenced by the static variable is determined with the *DominatesLeast* function, which analyses a set of reference contexts and returns the lowest in the hierarchy according to the *Dominates* relation. By taking the lowest possible reference context, the possibility of an error is maximised, which is essential to maintain a sound analysis.

The corresponding function in the formalisation to check the static variables in an environment is the *mSafeEnvStatic* function, which takes an environment *env*, a command *com*, and the *SCJ-mSafe* program p being checked, and returns a set of possible memory-safety violations. The function below is slightly different to the rule above in the sense that it takes a command as a parameter, and returns a set of possible violations as opposed to a boolean result. This is because the rules presented here demonstrate what must be true for a component to be memory safe, whereas the corresponding functions in the formalisation return information on the specific violations for a program. An empty set of violations characterises safety. The automatic tool, for usability, reports the violations specified in our formalisation, rather than just a boolean result, as suggested by the inference rules.

$$\begin{array}{l}
mSafeEnvStatic : Env \times Com \times SCJmSafeProgram \rightarrowtail \mathbb{P}\ Violation \\
\hline
\forall\, env : Env;\ com : Com;\ p : SCJmSafeProgram \\
\quad \bullet\ \exists\, rel : ExprShareRelation;\ ref : ExprRefSet \mid env = (rel, ref) \\
\qquad \bullet\ mSafeEnvStatic(env, com, p) = \\
\qquad\quad \{e1 : \operatorname{dom} ref;\ v : Violation \\
\qquad\qquad \mid e1 \in GetStaticVars\ p \\
\qquad\qquad \wedge\ (Dominates\_least(ref\ e1), IMem) \notin Dominates^{*} \\
\qquad\qquad \wedge\ v.com = com \\
\qquad\qquad \wedge\ v.rc1 = Dominates\_least(ref\ e1) \\
\qquad\qquad \wedge\ v.e1 = e1 \\
\qquad\qquad \wedge\ v.rc2 = IMem \\
\qquad\qquad\quad \bullet\ v\}
\end{array}$$

The function states that there must exist an expression share relation (*rel*) and expression reference set (*ref*) such that the environment is equal to the (*rel*, *ref*) pair. The result of the function is then a set of all violations found in the reference set characterised by an expression $e1$, which is in the set of static variables determined by the syntactic function *GetStaticVars*, where a mapping from the lowest possible reference context associated with the expression $e1$ to the immortal memory area *IMem* is not in the reflexive transitive closure of the *Dominates* relation. The formalisation here is the inverse of the hypothesis of the rule, that is the mapping is not in *Dominates* as opposed to must be in *Domiantes*, because the rule is specifying what must be true to guarantee safety, and the formalisation is specifying what must be true for a violation v to occur.

The *Violation* schema presented below defines a memory-safety violation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Violation}} \\
\hline
\mathit{com} : \mathit{Com} \\
e1 : \mathit{LExpr} \\
\mathit{rc}1 : \mathit{RefCon} \\
\mathit{rc}2 : \mathit{RefCon} \\
\hline
(\mathit{rc}2, \mathit{rc}1) \notin \mathit{Dominates}^{*} \\
\hline
\end{array}$$

The schema contains the *SCJ-mSafe* command *com* that caused the changes to the environment that introduced the possible violation. The expression $e1$ and reference context $rc1$ record the expression and associated reference context of the object that has caused the violation. The reference context $rc2$ records the reference context used as a comparison to $rc1$ to detect the violation; for example, if $e1$ has been identified as the expression that introduces an error because it is a field of another object and creates a downward reference, the reference context $rc2$ records the context of the containing object. If $e1$ is a local variable, $rc2$ records the reference context in which it was declared. Finally, if $e1$ is a static variable, $rc2$ is the immortal memory *IMem*.

Object fields Expressions that reference objects that are fields of other objects are checked against the reference context of the containing object.

The second hypothesis states that for every possible pair of expressions in the reference set (le_1 and le_2) such that the second expression is a field of the first, the worst-case mapping from the lowest possible reference context of the field to the highest possible reference context of the containing object is in the reflexive transitive closure of the *Dominates* relation. More specifically, this states that all fields of an object must reside in reference contexts that dominate the reference context of the containing object. The highest possible reference context of the containing object is determined with the *DominatesTop* function, which returns the highest reference context in a set. The highest reference context of the containing object and the lowest of the field is the worst case.

The corresponding function in the formalisation to check the object fields in an environment is the *mSafeEnvFields* function, which takes an environment *env*, a command *com*, and a set of local variables *localVars*, and returns a set of possible violations.

$$
\begin{array}{l}
mSafeEnvFields : Env \times Com \times \mathbb{P}\, LExpr \pfun \mathbb{P}\, Violation \\
\hline
\forall\, env : Env;\ com : Com;\ localVars : \mathbb{P}\, LExpr \\
\quad \bullet \exists\, rel : ExprShareRelation;\ ref : ExprRefSet \mid env = (rel, ref) \\
\qquad \bullet\ mSafeEnvFields(env, com) = \\
\qquad\quad \{e1, e2 : \mathrm{dom}\, ref;\ v : Violation \\
\qquad\qquad \mid FieldOf(e1, e2) = True \\
\qquad\qquad \land\ e2 \notin localVars \\
\qquad\qquad \land\ (Dominates\_least(ref\ e1), Dominates\_top(ref\ e2)) \notin Dominates^{*} \\
\qquad\qquad \land\ v.com = com \\
\qquad\qquad \land\ v.e1 = e1 \\
\qquad\qquad \land\ v.rc1 = Dominates\_least(ref\ e1) \\
\qquad\qquad \land\ v.rc2 = Dominates\_top(ref\ e2) \\
\qquad\qquad\quad \bullet\ v\}
\end{array}
$$

The function analyses all expressions in the expression reference set (*ref*) of the environment. For every possible pair of expressions $e1$ and $e2$ in the domain of the expression reference set, such that $e1$ is a field of $e2$ and $e2$ is not a local variable, there must be a mapping from the lowest possible reference context of $e1$ to the highest possible reference context of $e2$ in the reflexive transitive closure of the *Dominates* relation to be safe.

More specifically, if an expression $e1$ is a field of the object referenced by $e2$, the reference context of the field must dominate the reference context of the containing object, as otherwise this would be a downward reference. If the mapping is not in the *Dominates* relation, the violation v is returned, and the corresponding information about the error is recorded in the components of v.

Local variables When analysing *SCJ-mSafe* components that have their own local scope, the local variables declared are analysed based on the reference context in which they were defined, as opposed to the reference context of the containing object.

The third hypothesis states that for every possible expression in the reference set (le_1) such that the expression is a local variable, the worst-case mapping from the lowest possible reference context of the local variable to the current reference context is in the reflexive transitive closure of the *Dominates* relation. More specifically, this states that all local variables declared whilst analysing a particular component of the *SCJ-mSafe* program must reside in reference contexts that dominate the current reference context of the component being analysed. This facilitates the checking of local variables in the environment before they go out of scope.

The corresponding function in the formalisation to check the object fields in an environment is the *mSafeEnvLocal* function, which takes an environment *env*, a command *com*, a set of local variables *localVariables*, and the current reference context *rc*, and returns a set of possible violations.

$$mSafeEnvLocal : Env \times Com \times \mathbb{P}\, LExpr \times RefCon \rightarrow\!\!\!\!\!+\ \mathbb{P}\, Violation$$

```
∀ $env : Env;\ com : Com;\ localVars : \mathbb{P}\, LExpr;\ rc : RefCon$
  • $\exists\, rel : ExprShareRelation;\ ref : ExprRefSet \mid env = (rel, ref)$
    • $mSafeEnvLocal(env, com, localVars, rc) =$
      $\{e1 : \mathrm{dom}\, ref;\ v : Violation$
        $\mid e1 \in localVars$
        $\land (Dominates\_least(ref\ e1), rc) \notin Dominates^*$
        $\land v.com = com$
        $\land v.e1 = e1$
        $\land v.rc1 = Dominates\_least(ref\ e1)$
        $\land v.rc2 = rc$
          $\bullet\ v\}$
```

The expressions in *ref* being analysed must belong to the set of local variables *localVars*, and a mapping from the the lowest possible reference context of the expression $e1$ to the current reference context *rc* must exist in the reflexive transitive closure of the *Dominates* relation to be safe. If the mapping is not in *Dominates*, a violation *v* that contains the details of the error is returned.

Incomplete environment It is possible for environments to be incomplete when recursive data structures are used in a program. When checking the environment in this situation, there may be expressions in the environment that cannot be analysed in the ways described above. This happens when it has not be possible to determine the reference context of the containing object for an expression that is not a static or local variable.

Consider again the recursive data structure example described previously, but with an additional assignment on line 9.

```
1  Node n;
2  Node pt = new Node();
3  n = pt;
4  while(e) {
5    pt.next = new Node();
6    pt = pt.next;
7  }
8  ...
9  n.next.next.next = x;
```

This example demonstrates a case where the assignment at line 9 to field `n.next.next.next` cannot be checked against its containing object because the environment does not have information about the containing object `n.next.next`. The reference set of the environment after line 7 has been analysed in the example is shown below.

$$\{\ \ n \mapsto \{MMem\}, pt \mapsto \{MMem\}, n.next \mapsto \{MMem\}, pt.next \mapsto \{MMem\},\\ n.next.next \mapsto \{\}, pt.next.next \mapsto \{\}\ \ \}$$

Assuming that the object referenced by the variable `x` resides in the per-release memory area of a handler, the resulting environment after line 9 has been analysed is as follows.

$$\{\ \ n \mapsto \{MMem\}, pt \mapsto \{MMem\}, n.next \mapsto \{MMem\}, pt.next \mapsto \{MMem\},\\ n.next.next \mapsto \{\}, pt.next.next \mapsto \{\}, n.next.next.next.next \mapsto \{PRMem\ h\}\ \ \}$$

This assignment has introduced a memory-safety violation as the data structure referenced by `n` resides in mission memory, as do all of the subsequent elements of the data structure referenced by `n.next`, and `n.next.next`.... However, the assignment at line 9 creates a downward reference from the mission memory to the per-release memory area.

What is noteworthy is that the assignment changes a more-nested field than those identified by the calculation of the loop summary. The assignment is to a valid expression whose containing object is not captured in the environment.

Further analysis of the environment is required to establish the necessary information to detect the error, and maintain soundness. From the reference set shown above, it is possible to identify the downward reference by comparing the reference context of `n.next.next.next` against that of `n.next`, which is the object furthest into the data structure whose expression is a prefix of the one being checked and whose set of possible reference contexts is not empty.

It does not matter that the information about `n.next.next` is not known; the fact that the environment does not contain specific information about the set of possible reference contexts in which it may reside illustrates that there has been no specific assignment to the expression. Based on the analysis of recursive data structures as explained previously, the field `n.next` records the set of reference contexts in which all subsequent objects in the data structure reside. It is therefore safe to analyse `n.next.next.next` against `n.next` because the set of reference contexts in which `n.next.next` resides would be equal to that of `n.next` had a complete environment been created. If an assignment to `n.next.next` had occurred during the `...` on line 8 (or any expression that is it aliased with), the explicit set of reference contexts would be known, which would allow the checking of `n.next.next` against its containing object `n.next`, and also the result of the assignment on line for `n.next.next.next` against `n.next.next`.

The fourth hypothesis of the *mSafeEnv* rule is true if all expressions that have not been checked in an environment by the first three hypotheses are safe, according to the *Dominates* relation, when checked against the expression that has the longest prefix in the environment. Expressions that are not static or local variables, and do not have another expression in the environment such that it is a field of the second expression, are checked against the expression in the environment with the longest prefix. Fields of static variables and local variables are also be checked in this way.

The corresponding function in the formalisation to check incomplete environments is the *mSafeEnvIncomplete* function, which takes an environment *env*, a command *com*, a set of local variables *localVariables*, and the *SCJ-mSafe* program *p* being checked, and returns a set of possible violations.

$$
\begin{array}{|l}
\mathit{mSafeEnvIncomplete} : \mathit{Env} \times \mathit{Com} \times \mathbb{P}\,\mathit{LExpr} \\
\qquad \times \mathit{SCJmSafeProgram} \mathrel{+\!\!\!\rightarrow} \mathbb{P}\,\mathit{Violation} \\
\hline
\forall\, \mathit{env} : \mathit{Env};\ \mathit{com} : \mathit{Com};\ \mathit{localVars} : \mathbb{P}\,\mathit{LExpr};\ p : \mathit{SCJmSafeProgram} \\
\quad \bullet\ \exists\, \mathit{rel} : \mathit{ExprShareRelation};\ \mathit{ref} : \mathit{ExprRefSet} \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet\ \mathbf{let}\ \mathit{uncheckedExprs} == \operatorname{dom} \mathit{ref} \setminus \mathit{localVars} \cup \mathit{GetStaticVars}\ p \\
\qquad\quad \cup \bigcup\{ e : \operatorname{dom} \mathit{ref} \bullet \{ e1 : \operatorname{dom} \mathit{ref} \mid e \neq e1 \land \mathit{FieldOf}(e, e1) = \mathit{True} \bullet e \}\} \\
\qquad\quad \bullet\ \mathit{mSafeEnvIncomplete}(\mathit{env}, \mathit{com}, \mathit{localVars}, p) = \\
\qquad\qquad \{ e1 : \mathit{uncheckedExprs};\ e2 : \mathit{LExpr};\ v : \mathit{Violation} \\
\qquad\qquad\quad \mid e2 = \mathit{LongestPrefixOf}(\mathit{env}, e1) \\
\qquad\qquad\quad \land (\mathit{Dominates\_least}(\mathit{ref}\ e1), \mathit{Dominates\_top}(\mathit{ref}\ e2)) \notin \mathit{Dominates}^{*} \\
\qquad\qquad\quad \land v.\mathit{com} = \mathit{com} \\
\qquad\qquad\quad \land v.e1 = e1 \\
\qquad\qquad\quad \land v.\mathit{rc1} = \mathit{Dominates\_least}(\mathit{ref}\ e1) \\
\qquad\qquad\quad \land v.\mathit{rc2} = \mathit{Dominates\_top}(\mathit{ref}\ e2) \\
\qquad\qquad\qquad \bullet\ v \}
\end{array}
$$

The expressions that have not been successfully checked with the first three hypotheses are recorded in the *uncheckedExprs* set, which is defined as the domain of the reference set minus the local and static variables, and any expression in the environment that is a field of another expression in the environment.

The unchecked expressions are then checked in the same way as in the *mSafeEnvFields* function defined above, except that the expression used as a comparison ($e2$) is determined by the *LongestPrefixOf* function as opposed to the *FieldOf* function. The *LongestPrefixOf* function takes the environment and the expression $e1$ as its arguments and returns an expression from the environment that has the longest prefix of the expression $e1$ and whose set of possible reference contexts is not empty.

A prefix always exists in the environment as it is not possible to reference the field of an object that has not been declared, and if the outer-most containing object of the field has been declared, a prefix exists.

The function that defines the *mSafeEnv* rule as a whole in the formalisation is shown below; it has the same parameters as the *mSafeEnv* rule presented above plus the command *com* that is being analysed (as this is recorded in the violation if an error is found).

$$
\begin{array}{|l}
mSafeEnv : Env \times Com \times \mathbb{P}\, LExpr \times RefCon \\
\qquad \times\; SCJmSafeProgram \rightarrow\!\!\!\!\!\!+\; \mathbb{P}\, Violation \\
\hline
\forall\, env : Env;\; com : Com;\; localVars : \mathbb{P}\, LExpr;\; rc : RefCon; \\
\qquad p : SCJmSafeProgram \\
\quad \bullet\; mSafeEnv(env, com, localVars, rc, p) = \\
\qquad mSafeEnvStatic(env, com, p) \\
\qquad \cup\; mSafeEnvLocal(env, com, localVars, rc) \\
\qquad \cup\; mSafeEnvFields(env, com, localVars) \\
\qquad \cup\; mSafeEnvIncomplete(env, com, localVars, p)
\end{array}
$$

The *mSafeEnv* function returns a set of possible violations; the set of violations is calculated with the *mSafeEnvStatic*, *mSafeEnvLocal*, *mSafeEnvFields*, and *mSafeEnvIncomplete* functions.

4.6.3 Commands

SCJ-mSafe commands are the things that change the environment, and are therefore the components of *SCJ-mSafe* programs that can introduce memory-safety violations.

Commands are considered memory safe if their execution does not lead to an environment that contains a possible memory-safety violation.

The individual *mSafeCom* rules that define memory safety for each *SCJ-mSafe* command are described below. The parameters of each *mSafeCom* rule are an environment, the command being analysed, the set of local variables for the current *SCJ-mSafe* component being analysed, the current expression, the current reference context, and the *SCJ-mSafe* program being analysed.

Skip The *Skip* command is always safe as it has no behaviour, and the environment remains unchanged.

$$\frac{\mathit{True}}{\mathit{mSafeCom}(\mathit{env}, \mathit{Skip}, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

The conclusion to the *mSafeCom* rule when the command is *Skip*, is therefore always true.

Declaration As with the *Skip* command, declarations are always safe.

$$\frac{\mathit{True}}{\mathit{mSafeCom}(\mathit{env}, \mathit{Decl}\ d, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

As declarations can only add new expressions to the environment, and do not change the set of reference contexts an expression is associated with, they cannot introduce possible memory-safety violations; therefore, the conclusion of the rule is always true.

NewInstance As mentioned in the *CalcEnv* functions previously, the *NewInstance* command is one of the most important commands as it creates new objects and has the potential to introduce memory-safety violations.

$$\frac{\mathit{mSafeEnv}(\mathit{CalcEnvCom}(\mathit{env}, \mathit{NewInstance}(\mathit{nI}), \mathit{cexpr}, \mathit{rc}, p), \mathit{localVars}, \mathit{rc}, p)}{\mathit{mSafeCom}(\mathit{env}, \mathit{NewInstance}(\mathit{nI}), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

The rule has one hypothesis, which states that the resulting environment after the new instance command has been analysed must be memory safe according to *mSafeEnv*.

Assignment The assignment command is the other important command that can introduce memory-safety violations; the rule that allows us to prove memory-safety of assignments is shown below.

$$\frac{\mathit{mSafeEnv}(\mathit{CalcEnvCom}(\mathit{env}, \mathit{Asgn}(e1, e2), \mathit{cexpr}, \mathit{rc}, p), \mathit{localVars}, \mathit{rc}, p)}{\mathit{mSafeCom}(\mathit{env}, \mathit{Asgn}(e1, e2), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

As with new instance commands, the one and only hypothesis for the memory-safety of assignments states that the resulting environment after the assignment has been analysed must be memory safe according to *mSafeEnv*.

Sequence The rule for the sequence command has two hypotheses: the first states that the first command in the sequence must be memory safe, and the second states that the second command in the sequence must be memory safe.

$$\frac{\begin{array}{c}\mathit{mSafeCom}(\mathit{env}, c1, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)\\ \mathit{mSafeCom}(\mathit{CalcEnvCom}(\mathit{env}, c1, \mathit{cexpr}, \mathit{rc}, p), c2, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)\end{array}}{\mathit{mSafeCom}(\mathit{env}, \mathit{Seq}(c1, c2), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

The first hypothesis is true if the *mSafeCom* function applied to the first command in the sequence ($c1$) is also true; the environment passed to the function is the original environment *env*. The second hypothesis is true if the *mSafeCom* function applied to the second command in the sequence ($c2$) is also true; the environment passed to the function is the environment *env* updated with the effects of $c1$, which are calculated with the *CalcEnvCom* function.

If The rule for conditional statements is true if both the true and false branches of the conditional are also memory safe.

$$\frac{\begin{array}{c}\mathit{mSafeCom}(\mathit{env}, c1, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)\\ \mathit{mSafeCom}(\mathit{env}, c2, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)\end{array}}{\mathit{mSafeCom}(\mathit{env}, \mathit{If}(e, c1, c2), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

The first hypothesis is true if the true branch of the conditional statement ($c1$) is memory safe according to the *mSafeCom* function. The second hypothesis is true if the false branch of the conditional statement ($c2$) is also safe according to *mSafeCom*.

Both commands are analysed in the same environment *env*, as only one path executes at run-time, and it does not make sense to analyse one in the resulting environment of the other, like in the sequence command.

Switch The rule for switch commands is similar to the rule for conditional statements.

$$\frac{\begin{array}{c} \mathit{mSafeCom}(\mathit{env}, \mathit{comseq}.1, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p) \\ \ldots \\ \mathit{mSafeCom}(\mathit{env}, \mathit{comseq}.n, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p) \end{array}}{\mathit{mSafeCom}(\mathit{env}, \mathit{Switch}(e, \mathit{comseq}), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

Switch statements are memory safe if all of the possible commands in the switch statement are also memory safe according to the *mSafeCom* function (*comseq*.1 ... *comseq*.*n*). As with conditional statements, all possible commands in the switch statement are analysed in the same environment.

For For loops are memory safe if the initialisation command of the loop is safe, and the body of the loop followed by the iteration command is also safe.

$$\frac{\begin{array}{c} \mathit{mSafeCom}(\mathit{env}, c1, \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p) \\ \mathit{mSafeCom}(\mathit{CalcEnvCom}(\mathit{env}, c1, \mathit{cexpr}, \mathit{rc}, p), \mathit{Seq}(c2, c3), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p) \end{array}}{\mathit{mSafeCom}(\mathit{env}, \mathit{For}(c1, e, c2, c3), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

The first hypothesis states that the initialisation command $c1$ of the for loop must be safe in the environment *env* according to the *mSafeCom* function. The second hypothesis states that the sequence $\mathit{Seq}(c2, c3)$, which is the body of the loop followed by the iteration command must be safe in the resulting environment after the command $c1$ has been analysed, which is calculated by the *CalcEnvCom* function, according to the *mSafeCom* function.

Method call Method calls are memory safe if all of the possible methods that match the signature of the method call are safe.

$$\frac{\begin{array}{l} \forall\, \mathit{method} : \mathit{GetMethodsFromSigs}(\mathit{mc.methods}, p) \\ \qquad \bullet\ \mathit{mSafeEnv}(\mathit{env}', \mathit{localVars} \cup \mathit{m.localVars}, \mathit{rc}, p) \end{array}}{\mathit{mSafeCom}(\mathit{env}, \mathit{MethodCall}(\mathit{mc}), \mathit{localVars}, \mathit{cexpr}, \mathit{rc}, p)}$$

where
$\mathit{env}' = \mathit{ApplyPossibleMethods}(m, \mathit{mc.args}, \mathit{env}, \mathit{cexpr}, \mathit{mc.le}, \mathit{rc}, p)$

The hypothesis of the rule states that for all methods that may be executed as a result of the method call, which are determined by the *GetMethodsFromSigs* function, each resulting environment env' that is calculated with the *ApplyPossibleMethods* function, must be memory safe according to the *mSafeEnv* function. All possible methods according to *GetMethodsFromSigs* are analysed individually with the *ApplyPossibleMethods* function (hence the singleton set $\{m\}$ as the first argument to the call) as this identifies the specific method, if any, that introduces an error.

The set of local variables passed as a parameter to the *mSafeEnv* function are the local variables of the current component being analysed (*localVars*) joined with the local variables introduced as a result of analysing the corresponding method m ($m.localVars$).

EnterPrivateMemory The rule for the *EnterPrivateMemory* command is similar to regular method call rule presented above.

$$\frac{\begin{array}{l}\forall\, method : GetMethodsFromSigs(mc.methods, p) \\ \quad \bullet\ mSafeEnv(env', localVars \cup m.localVars, LowerRC(rc), p)\end{array}}{mSafeCom(env, EnterPrivateMemory(mc), localVars, cexpr, rc, p)}$$

where
$env' = ApplyPossibleMethods(m, mc.args, env, cexpr, mc.le, LowerRC(rc), p)$

The difference here is that the reference context rc passed to the *ApplyPossibleMethods* function to calculate the resulting environment for the execution of a particular method m is first passed to the *LowerRC* function, which returns the next lowest reference context based on the current context rc.

The new, lower, reference context is passed as a parameter to the *mSafeEnv* function as it is the lower reference context in which the environment should be analysed.

ExecuteInAreaOf The rule for the *ExecuteInAreaOf* command again analyses all of the possible methods associated with the method call mc, but it also takes into account the set of possible reference contexts in which the object specified as the area in which the method is to be executed, resides.

$$\frac{\begin{array}{l}\forall\, method : GetMethodsFromSigs(mc.methods, p) \\ \quad \bullet \forall\, rc1 : RCsFromMRC(mrc, rc, env.ref, p) \\ \qquad\quad \bullet\ mSafeEnv(env', localVars \cup m.localVars, rc1, p)\end{array}}{mSafeCom(env, ExecuteInAreaOf(mrc, mc), localVars, cexpr, rc, p)}$$

where
$env' = ApplyPossibleMethods(m, mc.args, env, cexpr, mc.le, rc1, p)$

The hypothesis is true if all possible methods that match the criteria of the method call *mc* are safe when analysed in all possible reference contexts of the object specified as the area in which the method is to be executed. The updated environment env' is calculated based on the possible reference context $rc1$ of the specified object.

ExecuteInOuterArea The rule for the *ExecuteInOuterArea* command is very similar to the *EnterPrivateMemory* command. The one difference is that the resulting environment calculated by the *ApplyPossibleMethods* function takes the immediate outer reference context that is calculated by the *RaiseRC* function as opposed to the immediately lower reference context.

GetMemoryArea The *GetMemoryArea* command is safe if the result of analysing the *GetMemoryArea* command does not produce an unsafe environment according to *mSafeEnv*.

$$\frac{mSafeEnv(CalcEnvCom(env, GetMemoryArea(gma), cexpr, rc, p), localVars, rc, p)}{mSafeCom(env, GetMemoryArea(gma), localVars, cexpr, rc, p)}$$

The hypothesis is true if the resulting environment calculated by the *CalcEnvCom* function when the *GetMemoryArea* command is analysed is memory safe according to *mSafeEnv*.

Try The rule for the *Try* command states that the try component of the try statement must be memory safe along with all of the commands in the catch statements and also the finally clause.

$$\frac{\begin{array}{c} mSafeCom(env, c1, localVars, cexpr, rc, p) \\ \forall\, com : \mathrm{ran}\, comseq \\ \bullet\; mSafeCom(env', com, localVars, cexpr, rc, p) \\ mSafeCom(env'', c2, localVars, cexpr, rc, p) \end{array}}{mSafeCom(env, Try(c1, eseq, comseq, c2), localVars, cexpr, rc, p)}$$

where
$env' = CalcEnvCom(env, c1, cexpr, rc, p)$
$env'' = DistEnvJoin(com : \mathrm{ran}\, comseq \bullet CalcEnvCom(env', com, cexpr, rc, p))$

The first hypothesis is true if the try statement $c1$ of the try command is memory safe according to the *mSafeCom* function. The second hypothesis is true if all catch commands in *comseq* are memory safe in the resulting environment env' that is calculated by analysing $c1$ with the *CalcEnvCom* function. Finally, the third hypothesis is true if the finally clause $c2$ of the try command is memory safe according to the *mSafeCom* function in the environment env'', which is the result of analysing the effect of all of the catch commands with the *CalcEnvCom* function and joining them with the *DistEnvJoin* function.

While The rule for the *While* command is true if the result of analysing the body of the while command is safe.

$$\frac{mSafeCom(env, com, localVars, cexpr, rc, p)}{mSafeCom(env, While(e, com), localVars, cexpr, rc, p)}$$

The hypothesis is true if the body of the while loop *com* is also memory safe according to *mSafeCom*.

DoWhile The rule for the *DoWhile* command is identical to the rule for the *While* command presented above, and so is omitted here.

Having presented the rules for the environment and *SCJ-mSafe* commands, the rules to check an entire *SCJ-mSafe* program are presented in a top-down approach next.

4.6.4 Overall *SCJ-mSafe* Program

An *SCJ-mSafe* program is only memory safe if all of its components are also memory safe. The rule to describe memory safety of an entire program is shown below.

$$\frac{\begin{array}{c} mSafeMethods(scjmsafe') \\ mSafeSafelet(env'', scjmsafe'.safelet, scjmsafe') \end{array}}{mSafe(scjmsafe)}$$

where
$scjmsafe' = BuildMethodProperties(scjmsafe)$
$env' = AddDecsToEnv(\varnothing, scjmsafe'.static)$
$env'' = DistEnvJoin(\{com : scjmsafe'.sInit \bullet CalcEnvCom(env', com, null, IMem)\})$

The rule states that a program *scjmsafe* is memory safe if all of the generated method properties are safe and if the analysis of the safelet is memory safe.

The first hypothesis that must be true checks that all of the method properties of the *SCJ-mSafe* program are memory safe. Method properties are checked independently of their execution context as it is possible to detect memory-safety violations that occur in methods regardless of the calling context. This is achieved with *mSafeMethods* that takes the *SCJ-mSafe* program $scjmsafe'$ as its parameter, which is defined as the original *SCJ-mSafe* program *scjmsafe* but with the method properties included. The *mSafeMethods* definition is presented in the next section. The method properties are generated as specified by the *BuildMethodProperties* function that was described previously.

The second hypothesis that must be true checks that the analysis of the safelet is safe. This is achieved with the *mSafeSafelet* function, which takes an environment, a safelet, and the *SCJ-mSafe* program being analysed as its parameters. The environment passed to the function is env'', which is the result of adding the static variables ($scjmsafe'.static$) to an empty environment with the *AddDecsToEnv* function to get env', and the subsequent analysis of all the static variable initialisation commands ($scjmsafe'.sInit$) in the immortal memory area with the *CalcEnvCom* function.

4.6.5 Safelet

The rule for the safelet component is used by the rule for the overall program above. It defines what must be true in order to guarantee that the safelet is memory-safety violation free.

$$\frac{\begin{array}{c} mSafeCom(env, s.initializeApplication, LocalVars(s.initializeApplication), \\ Null, IMem, p) \\ mSafeCom(env', s.getSequencer, LocalVars(s.getSequencer), Null, IMem, p) \\ mSafeMissionSeq(env'', p.missionSeq, p) \end{array}}{mSafeSafelet(env, s, p)}$$

where
$env' = CalcEnvCom(env, s.initializeApplication, Null, IMem, p)$
$env'' = CalcEnvCom(env', s.getSequencer, Null, IMem, p)$

The safelet component is memory safe if the `initializeApplication` method, `getSequencer` method, and mission sequencer components are safe.

The first hypothesis is true if the *mSafeCom* function applied to the `initializeApplication` method (*s.initializeApplication*) in the immortal memory area (*IMem*) is safe. The parameters passed to the function are the environment *env*, the command to be analysed (*s.initializeApplication*), the local variables of the `initializeApplication` method, the current expression *Null* (as this is the top level of execution), the current reference context (*IMem*), and the *SCJ-mSafe* program p.

The second hypothesis is true if the *mSafeCom* function applied to the `getSequencer` method (*s.getSequencer*) in the immortal memory area (*IMem*) is safe. It uses the environment that results from the execution of the `initializeApplication` method (env'), and is calculated with the *CalcEnvCom* function.

Finally, the third hypothesis is true if the *mSafeMissionSeq* function, which checks to see if the mission sequencer component is memory safe, is safe. The environment env'' results from the execution of the `getSequencer` method on env', and is calculated with the *CalcEnvCom* function.

4.6.6 Mission sequencer

The rule for the mission sequencer component is used by the safelet rule above, and defines what must be true for the mission sequencer to be free of memory-safety violations.

$$\frac{\begin{array}{c} mSafeCom(env, mSeq.getNextMission, LocalVars(mSeq.getNextMission), \\ GetMissionSeqExpr(p, mSeq), MMem, p) \\ mSafeMissions(env', mSeq.missions, p) \end{array}}{mSafeMissionSeq(env, mSeq, p)}$$

where
$env' = CalcEnvCom(env, mSeq.getNextMission, LocalVars(ms.getNextMission),$
$GetMissionSeqExpr(p, mSeq), MMem, p)$

The rule states that a mission sequencer *mSeq* is memory safe if the `getNextMission` method is safe, and all of the missions are safe.

The first hypothesis is true if the *mSafeCom* function applied to the `getNextMission` method (*mSeq.getNextMission*) in the mission memory area (*MMem*) is safe. The parameters of the *mSafeCom* function are the environment *env*, the command to be checked (*mSeq.getNextMission*), the set of local variables defined in the command being analysed, the expression that references the mission sequencer object, which is obtained through the syntactic function *GetMissionSeqExpr*, the current reference context (*MMem*), and the *SCJ-mSafe* program *p*.

The second hypothesis is true if the *mSafeMissions* function applied to the missions of the mission sequencer (*mSeq.missions*) are memory safe. The environment *env′* in which the missions are checked is the result of executing the `getNextMission` method in *env*, and is calculated with the *CalcEnvCom* function.

4.6.7 Missions

The rule that can be used to prove memory safety of a set of missions is the *mSafeMissions* rule, which is used by the rule for mission sequencers above.

$$\frac{\begin{array}{l} \exists\, m : missions \bullet mSafeMission(env, m, p) \\ \qquad\qquad \land\ mSafeMissions(env', missions \setminus \{m\}, p) \end{array}}{mSafeMissions(env, missions, p)}$$

where
$env' = CalcEnvMission(env, m, (GetMissionExpr(p, m)), p)$

The hypothesis is true if a mission *m* that belongs to the set *missions* is safe according to the *mSafeMission* function, and if all other missions that are not equal to *m* are also safe according to the same *mSafeMissions* function, but in an environment *env′*, which takes into account the behaviour of the mission m, and is calculated with the *CalcEnvMission* function.

In the hypothesis above, the application of *mSafeMissions* to the remaining missions in the set *missions* will eventually be empty, therefore the following rule is required to define that an empty set of missions is always safe.

$$\frac{\mathit{True}}{\mathit{mSafeMissions}(\mathit{env}, \varnothing, p)}$$

The *mSafeMission* function used in the definitions above states what must be true for a single mission to be memory safe.

$$\frac{\begin{array}{c}\mathit{mSafeCom}(\mathit{env}, m.\mathit{initialize}, \mathit{LocalVars}(m.\mathit{initialize}), \mathit{mExpr}, \mathit{MMem}, p)\\ \mathit{mSafeHandlers}(\mathit{env}', \mathit{GetHandlers}(p, m.\mathit{handlers}), p)\\ \mathit{mSafeCom}(\mathit{env}'', m.\mathit{cleanUp}, \mathit{LocalVars}(m.\mathit{cleanUp}), \mathit{mExpr}, \mathit{MMem}, p)\end{array}}{\mathit{mSafeMission}(\mathit{env}, m)}$$

where
$\mathit{mExpr} = \mathit{GetMissionExpr}(p, m)$
$\mathit{env}' = \mathit{CalcEnvCom}(\mathit{env}, m.\mathit{initialize}, \mathit{mExpr}, \mathit{MMem}, p)$
$\mathit{env}'' = \mathit{CalcEnvHandlers}(\mathit{env}', \mathit{GetHandlers}(p, m.\mathit{handlers}), p)$

The rule states that a mission is memory safe if the `initialize` method is safe, the execution of the mission's handlers is safe, and the `cleanUp` method is safe.

The first hypothesis is true if the mission's `initialize` method is safe according to the *mSafeCom* function. The parameters passed to the *mSafeCom* function are the environment *env*, the initialize method ($m.\mathit{initialize}$), the local variables declared during the initialize method, the expression that references the mission object (*mExpr*), the current reference context (*MMem*), and the *SCJ-mSafe* program being analysed (p).

The second hypothesis is true if the *mSafeHandlers* function is true, which checks the safety of all the handlers associated with the mission ($m.\mathit{handlers}$). The environment used to analyse the handlers (env') is the result of analysing the `initialize` method in the original environment *env*; this is calculated with the *CalcEnvCom* function.

Finally, the third hypothesis is true if the `cleanUp` method is safe according to the *mSafeCom* function. The environment used to analyse the `cleanUp` method (env'') is the result of applying *CalcEnvHandlers* to the set of handlers in the environment env'.

4.6.8 Handlers

The rule that can be used to prove memory safety of a set of handlers is the *mSafeHandlers* rule, which is used by the rule for missions above.

$$\frac{\begin{array}{c}\exists\, h : \mathit{handlers} \bullet \mathit{mSafeHandler}(\mathit{env}, h, \mathit{GetHandlerExpr}(p, h, m), p) \\ \wedge\ \mathit{mSafeHandlers}(\mathit{env}, m, \mathit{handlers} \setminus \{h\}, p)\end{array}}{\mathit{mSafeHandlers}(\mathit{env}, m, \mathit{handlers}, p)}$$

where
$\mathit{env}' = \mathit{CalcEnvHandler}(\mathit{env}, h, (\mathit{GetHandlerExpr}(p, h, m)), p)$

In order for the rule to be true, the set of handlers (*handlers*) must be memory safe. The hypothesis is true if a handler h that belongs to *handlers* is safe according to the *mSafeHandler* function, and all other handlers that are not equal to h are also safe according to the same *mSafeHandlers* function, but in an environment env', which takes into account the behaviour of the handler h, and is calculated with the *CalcEnvHandler* function.

In the hypothesis above, the application of *mSafeHandlers* to the remaining handlers in the set *handlers* will eventually be empty, therefore the following rule is required to define that an empty set of handlers is always safe.

$$\frac{\mathit{True}}{\mathit{mSafeHandlers}(\mathit{env}, m, \varnothing, p)}$$

The *mSafeHandler* rule used above states what must be true for a single handler to be memory safe.

$$\frac{\mathit{mSafeCom}(\mathit{env}, h.\mathit{handleEvent}, \mathit{LocalVars}(h.\mathit{handleEvent}), \mathit{hExpr}, \mathit{PRMem}\ h, p)}{\mathit{mSafeHandler}(\mathit{env}, h, \mathit{hExpr}, p)}$$

where
$\mathit{hExpr} = \mathit{GetHandlerExpr}(p, h, m)$

The rule states that a handler is memory safe if the `handleEvent` method is safe. The hypothesis is true if the handler's `handleEvent` method is safe according to the *mSafeCom* function. The parameters passed to the *mSafeCom* function are the environment *env*, the `handleEvent` method (*h.handleEvent*), the local variables declared during the `handleEvent` method, the expression that references the handler object ($hExpr$), the current reference context, which is the per-release memory area of the handler being analysed ($PRMem\ h$), and the *SCJ-mSafe* program being analysed (p).

The above memory-safety inference rules define what must be true for an *SCJ-mSafe* program to be memory safe. When applied to a program, if all hypotheses of all rules are true, then the program does not contain any possible memory-safety violations. If, however, one or more of the hypotheses are found to be false during the analysis, then that particular component may introduce a memory-safety violation.

By checking the environment after each command, it is possible to determine the exact location of a possible violation.

4.7 Checking method properties for memory-safety violations

This section describes how the checking technique establishes whether a possible memory-safety violation may occur based on the information recorded in method properties.

Like the *Dominates* relation for reference contexts defined previously, the *MRCDominates* relation describes what it means for relations between meta-reference contexts to be safe.

$$MRCDominates : MetaRefCon \leftrightarrow MetaRefCon$$

$$\begin{aligned}
MRCDominates = \{&e : LExpr;\ mrc : MetaRefCon \bullet (Erc\ e \mapsto mrc)\}\\
&\cup \{x : \mathbb{N} \bullet (CurrentPlus\ x \mapsto CurrentPlus(x-1))\}\\
&\cup \{(CurrentPlus\ 0 \mapsto Current)\}\\
&\cup \{(Current \mapsto CurrentPrivate\ 0)\}\\
&\cup \{x : \mathbb{N} \bullet (CurrentPrivate\ x \mapsto CurrentPrivate(x+1))\}\\
&\cup \{rcs1, rcs2 : \mathbb{P}\ RefCon\\
&\qquad \mid Dominates\_least\ rcs1 \mapsto Dominates\_top\ rcs2 \in Dominates^*\\
&\qquad\quad \bullet (Rcs\ rcs1 \mapsto Rcs\ rcs2)\}
\end{aligned}$$

The first part of the above definition states that any Erc meta-reference context dominates all other meta-reference contexts. In other words, if the location of an object is dependent on that of another expression that is not in the scope of the method properties being checked, then no decision can be made as to whether or not it is memory safe. Therefore the Erc meta-reference context dominates all others as the possibility of a memory-safety violation cannot be checked until applied to an environment.

The next set of relations state that all *CurrentPlus*(x) meta-reference contexts dominate those with a lower nesting level (*CurrentPlus*$(x-1)$). For example, if an object resides in *CurrentPlus*(2) (three meta-reference contexts above the *Current* context), then it dominates another meta-reference context *CurrentPlus*(1), which resides only two meta-reference context above the *Current* context.

The third part of the definition states that *CurrentPlus*(0) dominates *Current*, which is true as *CurrentPlus* meta-reference contexts represent reference contexts that are higher than the current reference context in the hierarchy. The next two parts of the definition state that *Current* dominates *CurrentPrivate*(0), which is the first private meta-reference context below *Current*, and that a meta-reference context *CurrentPrivate*(x) dominates a more nested context *CurrentPrivate*$(x+1)$. This is because *CurrentPrivate* meta-reference contexts represent reference contexts that are more-nested than the current reference context; that is, they are in a lower memory area.

The final part of the definition states that for two *Rcs* meta-reference contexts with possible sets of reference contexts $rcs1$ and $rcs2$ respectively, *Rcs* $rcs1$ dominates *Rcs* $rcs2$ if, and only if, the lowest possible reference context of $rcs1$ dominates the highest possible reference context of $rcs2$ according to the reflexive transitive closure of *Dominates*.

Using the *MRCDominates* relation above, it is possible to check method properties for possible memory-safety violations before the analysis of the program. Any memory-safety violations detected are independent of the execution, which therefore means the method in which the violation is detected is never safe, regardless of its calling point.

Method properties are checked in the same way as environments, that is, expressions in the method reference set that are fields of other expressions are compared against each other, and local variables are compared against the *Current* meta-reference context.

4.8 Rules for checking method properties

This section defines the memory-safety rules for method properties. As above, each rule has a set of hypotheses that must be true in order for conclusion of the rule to be true.

4.8.1 Method properties

The rule that defines what it means for a method's properties to be memory safe is *mSafeProperties* shown below.

$$\cfrac{\begin{array}{l}\land\ \forall (le_1, refSet_1), (le_2, refSet_2) : ref \mid \\ \quad FieldOf(le_1, le_2) \bullet \\ \quad \forall (mrc_1 : refSet_1), (mrc_2 : refSet_2) \bullet \\ \qquad mrc_1 \mapsto mrc_2 \in MRCDominates^* \\ \land\ \forall (le_1, refSet_1) : ref \bullet \\ \quad le_1 \in m.localVars \\ \quad \land\ \forall (mrc_1 : refSet_1) \bullet \\ \qquad mrc_1 \mapsto Current \in MRCDominates^* \\ \land\ \forall (le_1, refSet_1) : ref \mid \\ \quad le_1 \notin m.localVars \\ \quad \land\ (\neg\ \exists (le_2, refSet_2) : ref \bullet FieldOf(le_1, le_2)) \bullet \\ \qquad \exists (le_3, refSet_3) \mid le_3 = longestPrefixOf(properties, le_1) \bullet \\ \qquad\quad \forall (mrc_1 : refSet_1), (mrc_3 : refSet_3) \bullet \\ \qquad\qquad \land\ mrc_1 \mapsto mrc_3 \in MRCDominates^*\end{array}}{mSafeProperties(m, properties)}$$

where
$properties = (share, ref)$

The *mSafeProperties* rule is similar to the *mSafeEnv* rule in the sense that it checks all fields and local variables recorded in the method properties for possible memory-safety violations. It also checks expressions that are not checked by the first and second hypotheses, which check the fields and local variables respectively.

The first hypothesis, which checks the fields of objects, states that for every pair of entries in the method reference set of the method properties such that the expression of the second pair is a field of the expression of the first pair, there must be a mapping from all possible meta-reference contexts of the second expressions to all possible meta-reference contexts of the first expression in the reflexive transitive closure of the *MRCDominates* relation.

The second hypothesis, which checks local variables, states that for every entry in the method reference set of the method properties such that the entry's expression is a local variable, there must be a mapping from all possible meta-reference contexts associated with the expression to the *Current* meta-reference context in the reflexive transitive closure of the *MRCDominates* relation.

The third hypothesis, which checks expressions in the method properties that have not been checked by the first and second hypotheses, states that for every entry in the method reference set of the method properties such that the expression is not a local variable, and there is no other expression such that the first is a field of the second, there must be a mapping from all possible meta-reference contexts of the unchecked expression to all possible meta-reference contexts of the expression with the longest prefix in the reflexive transitive closure of the *MRCDominates* relation.

Unlike in the *mSafeEnv* rule, it may be possible that a longest prefix of an expression does not exist inside a method's properties, which means the expression remains unchecked until the properties are applied to an environment. This is because the containing object may be defined outside the scope of the method, and is not known until the properties are added to an environment.

4.8.2 All method properties

To ensure that all methods of an *SCJ-mSafe* program are memory safe, the *mSafeMethods* rule is defined below, which takes an *SCJ-mSafe* program as its parameter, and checks all of the methods for possible memory-safety violations.

$$\frac{\begin{array}{l}\forall\, m : p.safelet.methods \\ \quad \cup\; p.missionSeq.methods \\ \quad \cup\; \bigcup\{m : p.missions \bullet m.methods\} \\ \quad \cup\; \bigcup\{h : p.handlers \bullet h.methods\} \\ \quad \cup\; \bigcup\{c : p.classes \bullet c.methods\} \\ \quad \bullet\; mSafeProperties(m, m.properties)\end{array}}{mSafeMethods(p)}$$

The hypothesis of this rule states that for all methods in every *SCJ-mSafe* component in an *SCJ-mSafe* program p, the *mSafeProperties* rule must be true for each.

Once an SCJ program has been translated to *SCJ-mSafe*, and the method properties have been generated, the two rules above can be used to prove that no possible memory-safety violations exist inside methods when analysed independently of their calling context. The *mSafeMethods* definition is used by the *mSafe* rule described above, which checks the overall *SCJ-mSafe* program.

4.9 Final considerations

This chapter has presented the environment and method properties used to record the necessary information during the analysis of an *SCJ-mSafe* program in order to facilitate memory-safety checking. The corresponding formalisation of the environment and method properties has been presented here in Z; the full analysis formalisation can be found in Appendix E. This chapter has also described how the analysis maintains an up-to-date representation of the environment and method properties based on the semantics of individual *SCJ-mSafe* components.

This chapter has also presented the memory inference rules that characterise a technique to prove that an *SCJ-mSafe* program component is memory safe; the rules for memory-safe method properties have also been presented. The corresponding formalisation of the checking technique for environments and method properties has been presented in Z; the full checking formalisation can also be found in Appendix E.

The Z formalisation presented in this thesis has been developed and type checked using Z-Eves [38], however, no formal proof of its correctness has been carried out. In order to prove that the technique is correct, that is, to prove that a memory-safety violation in an SCJ program will be detected when translated to *SCJ-mSafe* and analysed with the technique defined in this chapter, a formal semantics of SCJ and its memory model must be defined. The definition of the SCJ memory model defined in the UTP discussed in Chapter 2 ([11]) is a good starting point for this; a complete semantics for SCJ that caters for its memory model does not exist yet.

The next chapter demonstrates how the overall technique presented in this thesis is applicable to SCJ programs. A number of specific test cases that produce memory-safety violations are presented, along with details of how the technique has been applied to several larger case studies.

Chapter 5

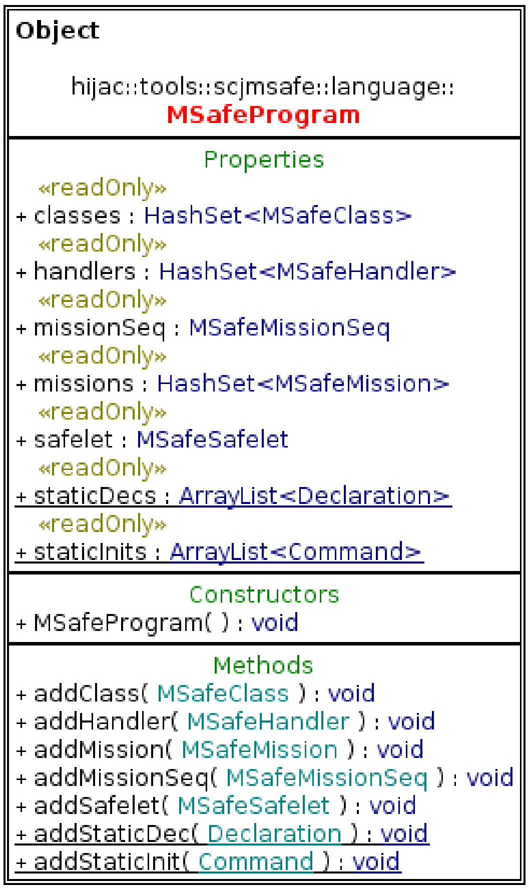
TransMSafe and examples

In order to evaluate the technique, a tool to check SCJ programs for possible memory-safety violations has been developed. The *TransMSafe* tool is described in Section 5.1, which implements the translation strategy defined in Chapter 3 and the checking strategy defined in Chapter 4. The tool, along with all of the examples described here and more, can be downloaded from [3].

A series of individual case studies that test specific features of the SCJ language are defined and applied to the tool. All of the individual examples illustrate how the features of SCJ can introduce possible memory-safety violations. Section 5.2 defines a series of examples that use the `enterPrivateMemory` and `executeInAreaOf` methods, and introduce possible memory-safety violations. It also illustrates how concurrency of Level 1 programs can introduce possibilities of memory-safety violations that do not occur at Level 0. Section 5.3 illustrates the applicability of the technique to larger case studies, and describes the possible memory-safety violations found when analysing them. Finally, Sections 5.4 and 5.5 evaluate the technique based on the results of the examples and draw some conclusions.

5.1 *TransMSafe*

The formalised translation from SCJ to *SCJ-mSafe* described in Chapter 3 has been fully automated in a tool called *TransMSafe* [3]. The tool is an extension to the tool described in [54]. The existing tool is implemented in Java and uses third-party utilities and libraries including the compiler tree API [2] to aid analysis and translation of SCJ programs; it is tailored for modifications and extensions.



```
Object

hijac::tools::scjmsafe::language::
MSafeProgram

Properties
  «readOnly»
+ classes : HashSet<MSafeClass>
  «readOnly»
+ handlers : HashSet<MSafeHandler>
  «readOnly»
+ missionSeq : MSafeMissionSeq
  «readOnly»
+ missions : HashSet<MSafeMission>
  «readOnly»
+ safelet : MSafeSafelet
  «readOnly»
+ staticDecs : ArrayList<Declaration>
  «readOnly»
+ staticInits : ArrayList<Command>

Constructors
+ MSafeProgram( ) : void

Methods
+ addClass( MSafeClass ) : void
+ addHandler( MSafeHandler ) : void
+ addMission( MSafeMission ) : void
+ addMissionSeq( MSafeMissionSeq ) : void
+ addSafelet( MSafeSafelet ) : void
+ addStaticDec( Declaration ) : void
+ addStaticInit( Command ) : void
```

Figure 5.1: Class diagram for the MSafeProgram class

The *SCJ-mSafe* model presented in Chapter 3 is encoded in Java in the implementation. Types defined in the model are represented as Java classes; components of definitions in the model are represented as fields of the class. Functions in the model are defined as methods in the implementation; the parameter types and return types correspond to the types in the model.

The overall architecture of the *TransMSafe* tool is split into three main parts. The first is the set of classes used to model *SCJ-mSafe* programs, the second is the set of translation classes that correspond to the *Translate* functions, and the third is the set of checking classes that correspond to the *CalcEnv* and *mSafe* functions. It is important to note that the implementation of the tool is an accurate representation of the formalisation so that examples can be applied to verify the technique is correct. To ensure this is the case, every component of the formalisation has a corresponding class or method in the implementation.

Figure 5.1 shows a class diagram of the `MSafeProgram` class, which represents the overall *SCJ-mSafe* program. As shown in the diagram, the class contains fields for each

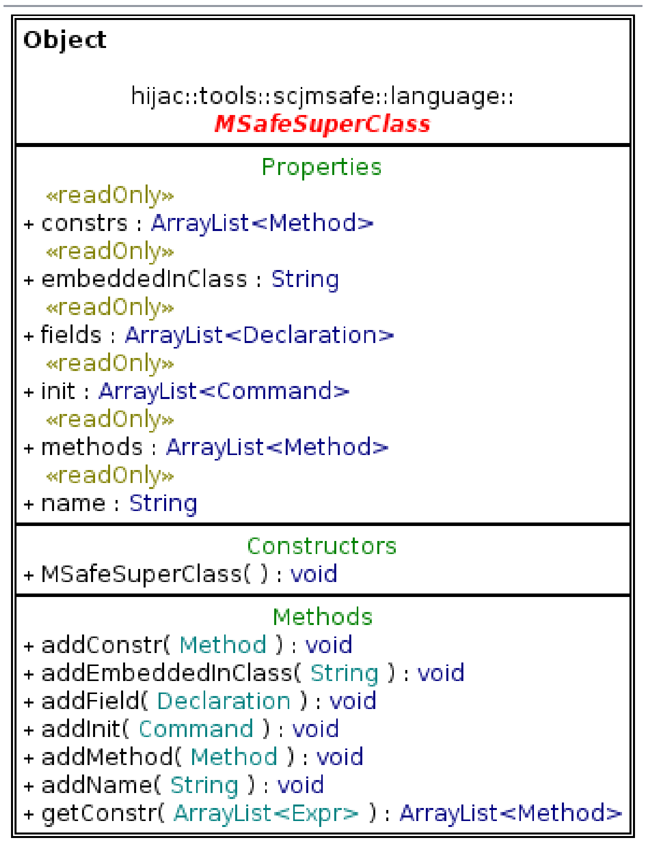

Object
hijac::tools::scjmsafe::language::
MSafeSuperClass
Properties
«readOnly»
+ constrs : ArrayList<Method>
«readOnly»
+ embeddedInClass : String
«readOnly»
+ fields : ArrayList<Declaration>
«readOnly»
+ init : ArrayList<Command>
«readOnly»
+ methods : ArrayList<Method>
«readOnly»
+ name : String
Constructors
+ MSafeSuperClass( ) : void
Methods
+ addConstr(Method) : void
+ addEmbeddedInClass(String) : void
+ addField(Declaration) : void
+ addInit(Command) : void
+ addMethod(Method) : void
+ addName(String) : void
+ getConstr(ArrayList<Expr>) : ArrayList<Method>

Figure 5.2: Class diagram for the MSafeSuperClass class

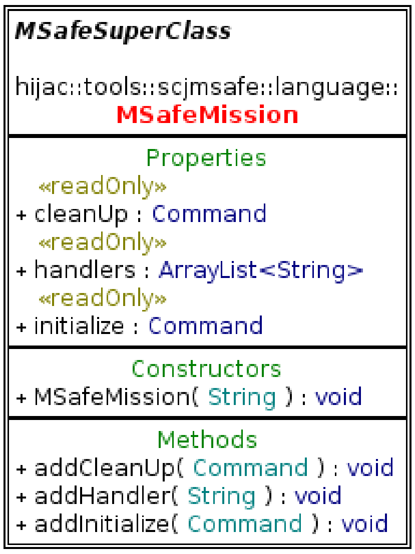

MSafeSuperClass
hijac::tools::scjmsafe::language::
MSafeMission
Properties
«readOnly»
+ cleanUp : Command
«readOnly»
+ handlers : ArrayList<String>
«readOnly»
+ initialize : Command
Constructors
+ MSafeMission(String) : void
Methods
+ addCleanUp(Command) : void
+ addHandler(String) : void
+ addInitialize(Command) : void

Figure 5.3: Class diagram for the MSafeMission class

component of the program, and has `add` methods for each.

All classes in *SCJ-mSafe*, including the paradigm classes, inherit from the `MSafeSuperClass` class, which is shown in Figure 5.2. The `MSafeSuperClass` contains components that are common to all classes in *SCJ-mSafe*, including the name, fields, field

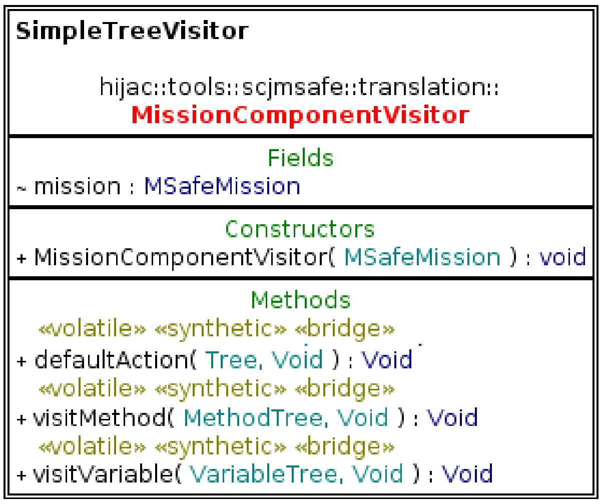

SimpleTreeVisitor
hijac::tools::scjmsafe::translation::
MissionComponentVisitor
Fields
~ mission : MSafeMission
Constructors
+ MissionComponentVisitor(MSafeMission) : void
Methods
«volatile» «synthetic» «bridge»
+ defaultAction(Tree, Void) : Void
«volatile» «synthetic» «bridge»
+ visitMethod(MethodTree, Void) : Void
«volatile» «synthetic» «bridge»
+ visitVariable(VariableTree, Void) : Void

Figure 5.4: Class diagram for the MissionComponentVisitor class

initialisation commands, constructors, and class methods.

Figure 5.3 shows the class diagram of the `MSafeMission` class, which represents *SCJ-mSafe* missions. The additional components here include the initialize method, clean-up method, and the associated handlers.

The *TransMSafe* tool uses the Java compiler tree API [2] to analyse the individual components of the input SCJ program and translate them into *SCJ-mSafe*. Each component has its own visitor, which is implemented as an extension of the Java `SimpleTreeVisitor` class. These visitors execute the corresponding `visitX` method, where `X` corresponds to the type of tree being analysed; for example, the `visitIf` method is used to identify and translate conditional statements.

These component visitors correspond to the individual *Translate* functions defined in Chapter 3. The *TranslateMission* function is implemented via the class `MissionComponentVisitor`, which analyses the variables and methods inside the class, and builds up the *SCJ-mSafe* mission (`MSafeMission`). Figure 5.4 shows the `MissionComponentVisitor` class diagram. The `visitMethod` and `visitVariable` methods are overridden from the `SimpleTreeVisitor` class to perform the translation.

The automatic translation has been successfully applied to several case studies and specific test cases that are discussed later in this chapter.

The formalised environment and method properties, and their corresponding calculation functions described in Chapter 4 have also been fully automated in the *TransMSafe* tool. The tool uses the translated SCJ program in *SCJ-mSafe* and checks for possible memory-safety violations.

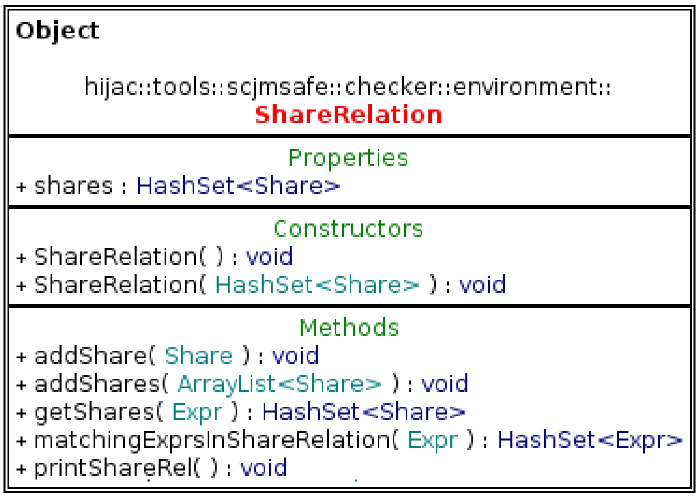

Object
hijac::tools::scjmsafe::checker::environment::
ShareRelation
Properties
+ shares : HashSet<Share>
Constructors
+ ShareRelation( ) : void
+ ShareRelation(HashSet<Share>) : void
Methods
+ addShare(Share) : void
+ addShares(ArrayList<Share>) : void
+ getShares(Expr) : HashSet<Share>
+ matchingExprsInShareRelation(Expr) : HashSet<Expr>
+ printShareRel( ) : void

Figure 5.5: Class diagram for the ShareRelation class

The models of the environment and method properties are translated into Java classes in the implementation. The functions used to generate method properties and update the environment during the analysis are defined as individual methods in Java. Like in the implementation of the translation, a class or method exists for each component of the model to ensure the implementation is a true representation of the formalisation.

Figures 5.5 and 5.6 show the expression share relation class `ShareRelation` and expression reference set class `RefSet` respectively. These correspond to the *ExprShareRelation* and *ExprRefSet* components of the environment.

As shown in Figure 5.5, the `ShareRelation` class has a field `shares`, which stores a set of `Share` objects. Each `Share` object has two fields that represent the left and right expressions of an individual mapping in an expression share relation.

The methods in the `ShareRelation` class correspond to the functions of the formalisation. For example, the `addShare` and `addShares` methods are used to add singular and multiple `Share` objects into the share relation; these correspond to the *ExprShareAdd* and *ExprShareAddSet* functions respectively. The `getShares` method is used to return all shares in the set `shares` that have the expression passed as a parameter in them; this gives the same functionality as the domain or range restriction functions in Z. The `matchingExprsInShareRelation` method is used to return a set of expressions that reference the same object based on the shares in the `shares` set. In Z, this is achieved with the domain and range restriction operators on the expression share relation.

Figure 5.6 shows that the `RefSet` class has a field `refSet`, which stores a set of `RefMapping` objects. Each `RefMapping` object has two fields that represent the left ex-

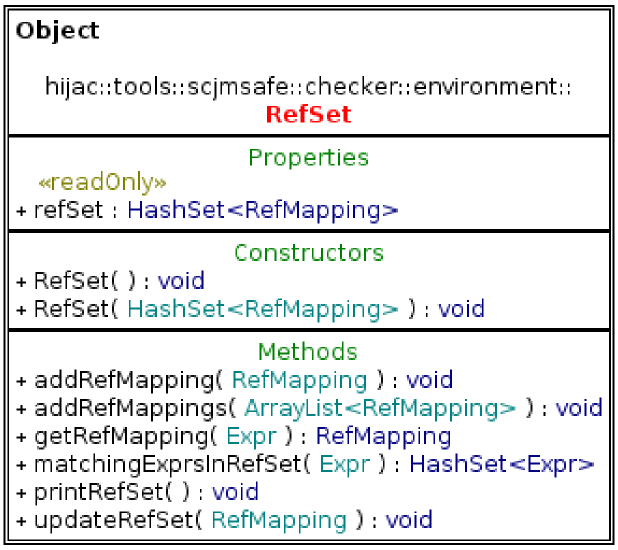

Object
hijac::tools::scjmsafe::checker::environment::
RefSet
Properties
«readOnly»
+ refSet : HashSet<RefMapping>
Constructors
+ RefSet( ) : void
+ RefSet(HashSet<RefMapping>) : void
Methods
+ addRefMapping(RefMapping) : void
+ addRefMappings(ArrayList<RefMapping>) : void
+ getRefMapping(Expr) : RefMapping
+ matchingExprsInRefSet(Expr) : HashSet<Expr>
+ printRefSet( ) : void
+ updateRefSet(RefMapping) : void

Figure 5.6: Class diagram for the RefSet class

pression and a set of possible reference contexts of an individual mapping in an expression reference set.

The methods in the `RefMapping` class also correspond to the functions of the formalisation. For example, the `addRefMapping` and `addRefMappings` methods are used to add `RefMapping` objects to the set `refSet`; these correspond to the *ExprRefAdd* and *ExprRefAddSet* functions respectively. The `getRefMapping` method takes an expression and returns the `RefMapping` object in the reference set that has the same expression. In Z, this is achieved by restricting the domain of the expression ref to the particular expression. The `updateRefSet` method takes a `RefMapping` object and updates the corresponding entry in the reference set with the same expression to include all reference contexts from the new and existing `RefMapping` object; this corresponds to the *ExprRefUpdate* function.

The automatic analysis of environments and method properties has been successfully applied to several case studies and specific test cases that have been automatically translated to *SCJ-mSafe*.

The formalised memory inference rules also described in Chapter 4 have been fully automated in the *TransMSafe* tool. Figure 5.7 shows the `SCJmSafeChecker` class which is responsible for the overall checking of the program.

As shown in Figure 5.7, the `SCJmSafeChecker` class contains fields that store references to the names and expressions of all the components of an *SCJ-mSafe* program; these fields store syntactic information generated from *SCJ-mSafe* programs, and correspond to the results of functions like *GetMissionExpr* and *GetHandlerExpr* in the formalisation. It

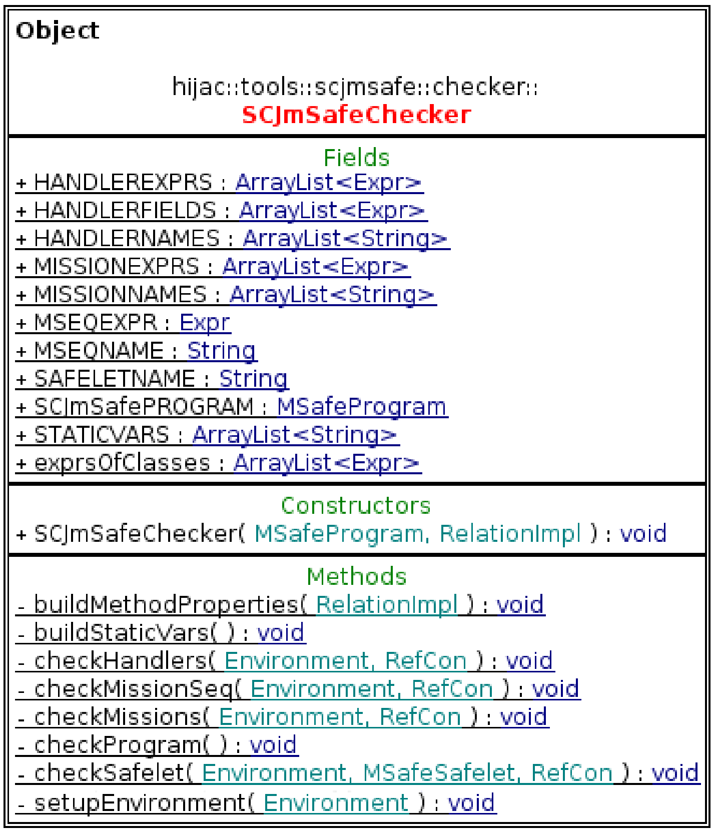

Object
hijac::tools::scjmsafe::checker::
SCJmSafeChecker
Fields
+ HANDLEREXPRS : ArrayList<Expr>
+ HANDLERFIELDS : ArrayList<Expr>
+ HANDLERNAMES : ArrayList<String>
+ MISSIONEXPRS : ArrayList<Expr>
+ MISSIONNAMES : ArrayList<String>
+ MSEQEXPR : Expr
+ MSEQNAME : String
+ SAFELETNAME : String
+ SCJmSafePROGRAM : MSafeProgram
+ STATICVARS : ArrayList<String>
+ exprsOfClasses : ArrayList<Expr>
Constructors
+ SCJmSafeChecker(MSafeProgram, RelationImpl) : void
Methods
- buildMethodProperties(RelationImpl) : void
- buildStaticVars( ) : void
- checkHandlers(Environment, RefCon) : void
- checkMissionSeq(Environment, RefCon) : void
- checkMissions(Environment, RefCon) : void
- checkProgram( ) : void
- checkSafelet(Environment, MSafeSafelet, RefCon) : void
- setupEnvironment(Environment) : void

Figure 5.7: Class diagram for the SCJmSafeChecker class

also includes individual methods to check the corresponding components of a program, for example `checkMissions` and `checkHandlers`, which correspond to the *mSafeMissions* and *mSafeHandlers* rules respectively. These are called by the `checkProgram` method, which implements the checking technique for the entire *SCJ-mSafe* program stored in the field `SCJmSafePROGRAM`.

5.2 Examples

This sections describes a number of specific test cases that illustrate how SCJ-specific components may introduce memory-safety violations. In particular, the examples include the use of the `enterPrivateMemory` method, the `executeInAreaOf` method, and concurrency.

5.2.1 Unit testing

During the development of the translation and checking tool, a series of specific test cases were used to ensure the behaviour that had been specified in the formalisation was correct. These examples included complex nested expressions and commands to ensure the translation was extracting side effects correctly, all the way to simple assignments during the checking phase to ensure that fields and equal expressions were being updated correctly.

During the translation from SCJ to *SCJ-mSafe*, it was important to test that each individual command and expression in SCJ was translated correctly into the corresponding *SCJ-mSafe* command or expression. Once complete, it was necessary to make the SCJ commands and expressions more complex, that is, embed commands and expressions inside other commands and expressions, and ensure that the translation extracted the side effects appropriately, and where necessary introduced the relevant new variables to ensure there were no embedded side effects in commands in *SCJ-mSafe*. It was also important to ensure that the SCJ paradigm classes were being identified correctly and translated into the dedicated *SCJ-mSafe* component.

During the checking of *SCJ-mSafe* programs, the unit tests ensured that the method properties and environment were updated correctly after every possible *SCJ-mSafe* command. For some commands, this was simple, as it was sufficient to check that no changes were being made; for example, the *Skip* command. Other commands, however, such as the assignment command, required more extensive testing. As identified in the previous chapter, the assignment command updates the environment in several different ways depending on the structure of the assignment itself. For example, an assignment to a field of an object where there exists an aliasing for that object in the environment is much more complicated than that of a simple primitive value assignment.

5.2.2 EnterPrivateMemory example

The `enterPrivateMemory` method executes a runnable object in a new temporary private memory area that is lower in the hierarchy than the current memory area. This introduces the possibility to create new objects in a lower memory area, and the potential to create references to these objects from reference variables that reside in higher memory areas.

For example, consider the simple example shown in Figure 5.8. In the handler's `handleAsyncEvent` method, a new object of type `A` is created and assigned to the local variable `a`; this object resides in the per-release memory area associated with the handler. The variable `a` is then passed as a parameter to the instantiation of the `MyRunnable` object referenced by the local variable `myRun`, which also resides in the per-release memory area.

The `MyRunnable` class constructor assigns the reference passed as a parameter to its own field called `aField`. At this point, the local variable `a` inside the handler's `handleAsyncEvent` method and the field `aField` inside the instance of `MyRunnable` referenced by `myRun` are aliased.

The final line of the `handleAsyncEvent` method calls the `enterPrivateMemory` method with the `myRun` variable as its parameter. This executes the `run` method of the `MyRunnable` class referenced by `myRun` in a new temporary private memory area associated with the handler. The `run` method creates a new object of type `Object` inside the temporary-private memory area and assigns it to the field `o` of the local field `aField`.

An extract of the reference set of the environment at the end of the execution of the `handleAsyncEvent` method is shown below.

$$
\begin{array}{l}
\{ \\
\quad \mathit{sequencer} \mapsto \{\mathit{IMem}\}, \\
\quad \mathit{sequencer.mission} \mapsto \{\mathit{MMem}\}, \\
\quad \mathit{sequencer.mission.handler} \mapsto \{\mathit{MMem}\}, \\
\quad \mathit{sequencer.mission.handler.a} \mapsto \{\mathit{PRMem}(\mathit{MyHandler})\}, \\
\quad \mathit{sequencer.mission.handler.a.o} \mapsto \{\mathit{TPMem}(\mathit{MyHandler}, 0)\}, \\
\quad \mathit{sequencer.mission.handler.myRun} \mapsto \{\mathit{PRMem}(\mathit{MyHandler})\}, \\
\quad \mathit{sequencer.mission.handler.myRun.aField} \mapsto \{\mathit{PRMem}(\mathit{MyHandler})\}, \\
\quad \mathit{sequencer.mission.handler.myRun.aField.o} \mapsto \{\mathit{TPMem}(\mathit{MyHandler}, 0)\}, \\
\quad \ldots \\
\}
\end{array}
$$

As shown from this extract, the field `sequencer.mission.handler.myRun.aField.o` (or `sequencer.mission.handler.a.o`) resides in the first temporary private memory area associated with the handler ($\mathit{TPMem}(\mathit{MyHandler}, 0)$). Its containing object `sequencer.mission.handler.myRun.aField` (or `sequencer.mission.handler.a`) resides in the per-release memory area of the handler ($\mathit{PRMem}(\mathit{MyHandler})$).

The execution of the `run` method in a new temporary private memory area through the `enterPrivateMemory` method call has introduced a possible memory-safety violation. The error is reported to the user as a possible violation because of the inherent nature of a worst-case analysis; whilst an error is guaranteed to be found, all reported errors may not be actual errors. To avoid confusion, the term possible is used as opposed to definite, for example. The resulting of analysing this example with the tool is the error message below that demonstrates the technique's ability to automatically find the potential memory violation.

```
public class MyHandler extends PeriodicEventHandler {

    ...

    public void handleAsyncEvent() {
        A a = new A();
        MyRunnable myRun = new MyRunnable(a);
        ManagedMemory.enterPrivateMemory(1, myRun);
    }

    class MyRunnable implements Runnable {
        A aField;

        public MyRunnable(A arg) {
            aField = arg;
        }

        public void run() {
            aField.o = new Object();
        }
    }
}
```

Figure 5.8: Example of possible memory-safety violation introduced by the enterPrivate-Memory method.

> POSSIBLE MEMORY SAFETY VIOLATION - The field 'sequencer.mission.handler.myRun.aField.field' may reference an object stored in 'TPMem(MyHandler, 0)' when its containing object 'sequencer.mission.handler.myRun.aField' may reside in 'PRMem(MyHandler)'

This is an important example as it illustrates how the passing of references to runnable objects may cause memory-safety violations when the corresponding `run` method is executed in a lower memory area. This type of error is not restricted to the per-release and temporary-private memory areas shown in the example above. It may also arise in the `initialize` method of a mission where an object that resides in mission memory is passed as a reference to a runnable object that executes in a temporary private memory area associated with the mission. Similarly, if an object that resides in a temporary private memory area is passed as a reference to a runnable object that executes in a more nested temporary private memory area, the same error may arise.

The two other main techniques that are capable of detecting memory-safety violations of Level 1 SCJ programs are the SCJChecker [45] and bytecode analysis [14] techniques; both of these techniques are capable of detecting this type of error.

In order for the SCJChecker to detect this error, it would be necessary to define annotations that specify which memory areas the `MyRunnable` and `A` classes must reside in, along with the memory area in which the runnable object's `run` method executes in. With these annotations, at the point of the new instantiation inside the `run` method, the SCJChecker is able to identify that the new object stored in a field of the `A` class does not reside in the same memory area in which objects of class `A` must reside.

The bytecode checker is able to detect the error using a similar method to the technique presented here. Neither the bytecode checking technique nor the technique presented here rely on annotations in order to check for possible memory-safety violations. In addition, neither technique restrict instances of the `MyRunnable` or `A` classes to a particular memory area.

5.2.3 ExecuteInAreaOf example

The `executeInAreaOf` method executes a runnable object in the memory area of a particular object. This introduces the possibility of creating new objects in memory areas other than the current memory area. Like in the example above, this may introduce the possibility of downward references.

For example, consider the example shown in Figure 5.9. In this example, the handler field `handlerField` is declared and instantiated in the mission memory area, because the handler object itself resides in the mission memory area. When the `handleAsyncEvent` method executes, the per-release memory area for the handler is entered. The local variable `data` is instantiated with a new object that resides in the per-release memory area. The local variable `data` is then passed to the constructor of the `MyRunnable` class on line 9 and is stored in the field `runField` of the object referenced by `myRun`.

When the runnable object referenced by `myRun` is passed as an argument to the `executeInAreaOf` method, the `run` method is executed in the same memory area as the first argument passed to the `executeInAreaOf` method. In this example, the first argument is the `handlerField` variable, which references an object that resides in mission memory; the `run` method is therefore executed in mission memory.

The `run` method assigns the local field `runField` to the handler field `handlerField`. As the local field `runField` references an object that was instantiated in the per-release memory area of the handler (line 8), and `handlerField` is a field of the handler object that resides in mission memory, a possible downward reference is introduced. The resulting

```java
public class MyHandler extends PeriodicEventHandler {

    Object handlerField = new Object();

    ...

    public void handleAsyncEvent() {
        Object data = new Object();
        MyRunnable myRun = new MyRunnable(data);
        ManagedMemory.executeInAreaOf(handlerField, myRun);
    }

    class MyRunnable implements Runnable {
        Object runField;

        public MyRunnable(Object arg) {
            runField = arg;
        }

        public void run() {
            handlerField = runField;
        }
    }
}
```

Figure 5.9: Example of possible memory-safety violation introduced by the executeInAreaOf method.

environment after the `handleAsyncEvent` method has executed is shown below.

$$\begin{array}{l}
\{\\
\quad sequencer \mapsto \{IMem\},\\
\quad sequencer.mission \mapsto \{MMem\},\\
\quad sequencer.mission.handler \mapsto \{MMem\},\\
\quad sequencer.mission.handler.handlerField \mapsto \{MMem, PRMem\ handler\},\\
\quad sequencer.mission.handler.data \mapsto \{PRMem\ handler\},\\
\quad sequencer.mission.handler.myRun \mapsto \{PRMem\ handler\},\\
\quad sequencer.mission.handler.myRun.runField \mapsto \{PRMem\ handler\},\\
\quad \ldots\\
\}
\end{array}$$

As shown in the environment above, *sequencer.mission.handler.handlerField* has two possible reference contexts in which the referenced object may reside. This is because the field is instantiated at the point of declaration when the handler object is created, and so may reside in the mission memory area (*MMem*), but also the later assignment as part of the `executeInAreaOf` method call introduced the possibility of *sequencer.mission.handler.handlerField* referencing an object that resides in the per-release memory area associated with the handler.

As *sequencer.mission.handler.handlerField* is a field of *sequencer.mission.handler*, it is checked against the location in which *sequencer.mission.handler* may reside. In this example, *sequencer.mission.handler* resides in mission memory (*MMem*). There is a possibility that *sequencer.mission.handler.handlerField* may reside in the per-release memory *PRMem handler*, therefore, a potential memory-safety violation is raised. The tool produces the following error message.

> POSSIBLE MEMORY SAFETY VIOLATION - The field 'sequencer.mission.handler.handlerField' may reference an object stored in 'PRMem(MyHandler)' when its containing object 'sequencer.mission.handler' may reside in 'MMem'

As in the previous example, this `executeInAreaOf` example illustrates how the passing of references to runnable objects that execute in different memory areas may introduce memory-safety violations. This example is useful to highlight two possible sources of errors in SCJ programs. The first is that despite the fact that the runnable object in this example is being executed in a memory area higher than the current memory area, it is still possible to introduce possible memory-safety violations with the `executeInAreaOf` method. The second is that fields of handlers cannot be instantiated or assigned new objects that are created in the per-release memory area, or more specifically during the `handleEvent` method of the handler. This is because handler fields must reside in the mission memory or higher, as handler objects themselves are created in the mission memory, and new objects instantiated inside the `handleEvent` method reside in the per-release memory area of the handler.

The `executeInOuterArea` method can also produce possible memory-safety violations in the same way as the `executeInAreaOf` method. The main difference between the two methods is that the `executeInOuterArea` method executes runnable objects in the immediately outer scope whilst the `executeInAreaOf` method executes them in the memory area of any object passed as an argument.

The SCJChecker would prevent this type of memory-safety violation occurring as it would not be possible to create a new instance of the object referenced by the handler field `handlerField` in the mission memory and then another referenced by the local variable `data` in the per-release memory area. This is because in order to facilitate checking, a scope annotation would be required on the class being instantiated, which would restrict

```
public class MyHandler1 extends PeriodicEventHandler {

    A sharedData;
    MemoryArea iMemRef;
    MemoryArea mMemRef;

    public MyHandler1(PriorityParameters priority,
       PeriodicParameters release, StorageParameters storage,
       A data, MemoryArea iMem, MemoryArea mMem) {
        super(priority, release, storage);
        sharedData = data;
        iMemRef = iMem;
        mMemRef = mMem;
    }

    public void handleAsyncEvent() {
        sharedData.entry = (A) iMemRef.newInstance(A.class);
        ...
        sharedData.entry = (A) mMemRef.newInstance(A.class);
    }
}
```

Figure 5.10: Example of possible memory-safety violation introduced through concurrency: MyHandler1.

objects of that type to a particular memory area. In order to cater for this type of example, where instances of a class resides in two different memory areas, class duplication would be required, which this technique does not require.

The bytecode analysis technique is not currently capable of handling the `executeInAreaOf` method, and therefore would not be able to handle this example. This restriction is not imposed through a limitation of the bytecode technique, and support could be added relatively easily.

5.2.4 Concurrency example

Level 1 SCJ programs introduce the possibility of concurrency. The previous chapter described how the technique handles concurrency. As an example, we consider the event handler shown in Figure 5.10, which is an implementation of the example presented in Figure 4.9. This event handler contains a local field `sharedData` that references an object stored in mission memory; a reference to this object is passed to all the handlers in the program so that data can be shared between them. The handler defines local fields that contain references to the immortal and mission memory areas respectively, which are also passed as arguments to the constructor.

During the `handleAsyncEvent` method of the handler, the field `entry` of the object

```
public class MyHandler2 extends PeriodicEventHandler {

    A sharedData;
    MemoryArea mMemRef;

    public MyHandler2(PriorityParameters priority,
       PeriodicParameters release, StorageParameters storage,
       A data, MemoryArea mMem) {
        super(priority, release, storage);
        sharedData = data;
        mMemRef = mMem;
    }

    public void handleAsyncEvent() {
        ...
        sharedData.entry.field = (A)
           mMemRef.newInstance(A.class);
        ...
    }
}
```

Figure 5.11: Example of possible memory-safety violation introduced through concurrency: MyHandler2.

referenced by the expression `sharedData` is instantiated twice; firstly in the immortal memory area, and secondly in the mission memory area. This is done with the `newInstance` command, which returns a reference to a newly created object in a particular memory area.

When considering the behaviour of this handler independently, all seems fine. This is because the object referenced by `sharedData` resides in mission memory, and the field `entry` either resides in the immortal memory area, which is an upward reference, or in the mission memory, which is the same memory area. The resulting environment after the `handleAsyncEvent` method has been analysed is shown below.

$$
\begin{array}{l}
\{\\
\quad \mathit{sequencer} \mapsto \{\mathit{IMem}\},\\
\quad \mathit{sequencer.mission} \mapsto \{\mathit{MMem}\},\\
\quad \mathit{sequencer.mission.handler1} \mapsto \{\mathit{MMem}\},\\
\quad \mathit{sequencer.mission.handler1.iMemRef} \mapsto \{\mathit{IMem}\},\\
\quad \mathit{sequencer.mission.handler1.mMemRef} \mapsto \{\mathit{MMem}\},\\
\quad \mathit{sequencer.mission.handler1.sharedData} \mapsto \{\mathit{MMem}\},\\
\quad \mathit{sequencer.mission.handler1.sharedData.entry} \mapsto \{\mathit{MMem}, \mathit{IMem}\},\\
\quad \ldots\\
\}
\end{array}
$$

In the environment above, no memory-safety violations may occur as all references are either in the same memory area or up the hierarchy.

The second handler in the program is shown in Figure 5.11. This handler also contains a field that references the shared data object stored in the mission memory area. During the execution of the `handleAsyncEvent` method of this handler, the field `sharedData.entry.field` is instantiated with a new object that resides in the mission memory area. When analysed on its own, this behaviour is fine because the object referenced by `sharedData` resides in the mission memory area, and so does the new object that is being created.

However, it may be true that these handlers execute concurrently, which introduces the possibility that the expression `sharedData.entry.field` references an object that resides in mission memory while its containing object referenced by `sharedData.entry` resides in the immortal memory area. This introduces a possible memory-safety violation as a downward reference exists from `sharedData.entry` to `sharedData.entry.field`. The resulting environment after both handlers have been analysed is shown below.

$$
\begin{array}{l}
\{\\
\quad sequencer \mapsto \{IMem\},\\
\quad sequencer.mission \mapsto \{MMem\},\\
\quad sequencer.mission.handler1 \mapsto \{MMem\},\\
\quad sequencer.mission.handler1.iMemRef \mapsto \{IMem\},\\
\quad sequencer.mission.handler1.mMemRef \mapsto \{MMem\},\\
\quad sequencer.mission.handler1.sharedData \mapsto \{MMem\},\\
\quad sequencer.mission.handler1.sharedData.entry \mapsto \{MMem, IMem\},\\
\quad sequencer.mission.handler1.sharedData.entry.field \mapsto \{MMem\},\\
\quad sequencer.mission.handler2 \mapsto \{MMem\},\\
\quad sequencer.mission.handler2.mMemRef \mapsto \{MMem\},\\
\quad sequencer.mission.handler2.sharedData \mapsto \{MMem\},\\
\quad sequencer.mission.handler2.sharedData.entry \mapsto \{MMem, IMem\},\\
\quad sequencer.mission.handler2.sharedData.entry.field \mapsto \{MMem\},\\
\quad \ldots\\
\}
\end{array}
$$

Without the history element of the environment, the fact that `sharedData.entry` ever referenced an object that resides in the immortal memory area would be lost, as it is overwritten with a reference to an object that resides in the mission memory later in the execution. During the analysis of the second handler, the tool outputs the following error message.

```
POSSIBLE MEMORY SAFETY VIOLATION - The field 'sequencer.mission.handler2.sharedData.entry.field' may reference an object stored in 'MMem' when its containing object 'sequencer.mission.handler2.sharedData.entry' may reside in 'IMem'
```

The history maintained in the environment makes it possible to detect errors that are only introduced through concurrent execution of handlers. It also allows the technique to analyse missions and handlers independently of the actual execution order in the program.

The SCJChecker is capable of handling concurrency because restrictions are put in place at the point of code annotation rather than program analysis. More specifically, it would not be possible to specify annotations for the example program presented here for the SCJChecker. This is because the instantiation of the two objects in the first handler's `handleAsyncEvent` method are in different memory areas, which is not possible in the SCJChecker as a necessary scope annotation would restrict instances of the class to a particular memory area. In order to get around this limitation, code duplication would be required.

The bytecode analysis technique is capable of handling Level 1 programs [14], however there is no description in published material that explains how the technique handles concurrency. Subsequently it is not clear how, if at all, the technique is able to handle examples such as this.

5.3 Case studies

The examples above describe specific cases that may produce memory-safety violations in a program. These small examples illustrate that the technique is capable of detecting violations, however the application of the technique to larger case studies is also interesting. Figure 5.12 shows a table with the data for the case studies and examples above run through the tool; the case studies are discussed individually below.

CDx One of the most interesting examples discussed in the SCJ literature is the collision detector simulator (CDx) [21]. The CDx is a flight collision detection algorithm that calculates the possible collisions of aircraft based on their position and movement, and is a benchmark for SCJ. This example is interesting because it makes use of the new SCJ-specific methods that have a direct impact on memory; the `executeInAreaOf` method

Example	**LOC**	**Analysis time (secs)**	**Expected errors**	**Errors found**	**False negatives**
ACCS	847	1.8	0	0	0
CDx	2,852	16.5	0	1	0
InOutParameter	169	1.3	2	1	0
Minepump	1,447	3.3	1	1	0
Pacemaker	780	2.5	0	0	0
Papabench	6,373	21.4	0	0	0
SCJChecker	148	1.2	0	0	0
Concurrency	138	2.1	1	1	0
EnterPrivateMemory	106	1.2	1	1	0
ExecuteInAreaOf	97	1.2	1	1	0

Figure 5.12: Table showing example and case study data.

is used, for example. Although this method is used, it is used safely, and therefore no memory-safety violations were expected when analysing the tool, as show in the fourth column of Figure 5.12. The tool is able to translate the CDx program, generate the method properties, and analyse the program successfully in an average of 16.5 seconds; the CDx has nearly 3k lines of code. These results were gathered from a system running Linux, with an Intel Core i5 650 processor at 3.20GHz, with 8GB RAM.

As shown from the figure, the output of the tool for the CDx example included a possible memory-safety violation, which was not expected. Upon closer inspection of the code, the error raised is a valid error that may occur during execution. The error occurs when an array object that is a handler field is updated to include a new object that is created during the `handleEvent` method of the handler. This is a genuine error because the handler field resides in the mission memory whilst the new object that is added to the array is defined in the per-release memory area. The containing array object therefore references an object that resides in a lower memory area. This error has previously gone undetected in the literature for other techniques. The tool raised no other errors, and therefore no false-negatives were raised.

ACCS Another example is the Automotive Cruise Controller System (ACCS) [49] presented in Chapter 3 that automatically monitors and maintains the speed of a vehicle. The implementation is a Level 1 program with a single mission and is made up from seven handlers that monitor the vehicle's gears, engine, brakes, throttle, levers, wheel shaft, and speed. The example does not include any interesting uses of SCJ-specific methods that affect memory, and therefore no errors were expected to arise when put through the tool.

This was confirmed when analysed, as no errors were returned; this also means that no false-negatives were raised during the analysis. The analysis took an average of 1.8 seconds to complete for 847 lines of code.

PapaBench and Pacemaker Two other examples analysed with the tool include the PapaBench implementation, which is a real-time benchmark adapted for SCJ [30], and an SCJ pacemaker implementation described in [44]. Once again, neither implementation includes any interesting uses of memory, and therefore there were no expected errors for either example. As expected, there were no possible memory-safety violations raised by the tool in either example. Once again, no false-negatives were raised.

The analysis times for the PapaBench and pacemaker examples were 21.4 and 2.5 seconds, on average, respectively. The PapaBench case study is the largest analysed here, and is made up of over 6k lines of code. The pacemaker example is significantly smaller at less than 1k lines of code.

SCJChecker example The SCJChecker technique includes an example in the literature that demonstrates the need for class duplication when annotating classes that are instantiated in different memory areas [45]. This example has been reworked to remove the annotations and class duplication and checked with the tool. No errors are raised and no false-negatives reported, which illustrates the ability to check SCJ programs without the need for class duplication or user-added annotations.

Other case studies The literature associated with the bytecode analysis technique describes several other SCJ examples, some of which include potential memory-safety violations [14]. These include a simple in-out parameter implementation that writes the values of an input to a string output. The output of the tool for this example includes a possible memory-safety violation as a static variable may reference an object that resides in the mission memory. Inspection of the code revealed that the error is a legitimate memory-safety violation as one of the static variables is assigned to point to a new object that is created in the mission's `initialize` method, which executes in the mission memory area. Objects referenced by static variables must reside in the immortal memory area. The bytecode analysis technique is also capable of detecting this error, and as such, the error was expected, as demonstrated in Figure 5.12.

The other expected error in this example comes from a buffer object that may be re-allocated in a temporary private memory area if the buffer length is exceeded. This re-allocation causes a downward reference, which is a memory-safety violation. The *TransMSafe* tool does not report this because a stub reference implementation is used; if a full reference implementation was used whilst analysing programs, then the *TransMSafe* tool would also raise the possible error. This is reflected in Figure 5.12 as the number of errors detected is less than the number of errors expected.

Another example used in the bytecode analysis technique is the textbook mine-pump example [9]. This version includes logging features and includes a engineered memory-safety violation for testing purposes. The output of the *TransMSafe* tool includes a possible memory-safety violation, which is the error deliberately introduced. Similarly to the CDx example, the error occurs because a new array entry is created in a memory area that is lower than that of the array object itself.

5.4 Evaluation

The *TransMSafe* tool described here provides validation that the formalised technique described in Chapters 3 and 4 is capable of detecting potential memory-safety violations in SCJ programs. This has been shown by the automatic translation and analysis of several case studies and a series of specific test cases that are known to generate memory-safety violations.

A number of lessons were learnt during the implementation of the tool. For example, in the simple assignment test case, it became clear that the formalisation was not correct. In the specific case where an assignment is of the form `a.b = x`, the existing expressions that are aliased with `a` were not being updated to include the new value of the field `b`.

The implementation of the checking technique also helped to identify the necessary steps to handle concurrency; for example, like in the concurrency example presented previously, it soon became clear that a naive approach of analysing handlers independently was not sufficient. Instead this lead to the history element of the environment, which facilitates the individual analysis of handlers whilst also dealing with concurrency.

The implementation of the translation and checking tool totals 13k lines of Java code and is an extension to the hiJaC tools described in [54]. The implementation is a proof-of-concept for the technique defined here, and it is inevitable that attempts to optimise the code will improve the translation and analysis times. Optimisation opportunities are discussed in more detail in the next chapter.

5.5 Final considerations

This chapter has described the *TransMSafe* tool that translates SCJ programs into *SCJ-mSafe* and checks for possible memory-safety violations. A number of specific examples that illustrate the ability to detect memory-safety violations have also been presented along with a selection of case studies. Finally, an evaluation of the tool and technique is given. The next chapter draws a conclusion on this thesis and describes possible future work.

Chapter 6

Conclusions and further work

This chapter summarises the work presented here and describes some possible extensions to the technique that could be completed as further work.

6.1 Summary

The original hypothesis of this work claimed that it was possible to produce an automatic static checking technique for valid Level 1 SCJ programs to identify possible unsafe uses of memory at the source-code level, without the need for additional user-added annotations. The previous chapters and examples described here demonstrate that this hypothesis is true.

This thesis has defined a memory-safety checking technique for SCJ programs. Several other techniques to verify memory safety of SCJ programs have been presented, however, currently the technique presented here is the only practical solution at the source-code level for Level 1 SCJ programs that maintains traceability, does not raise unnecessary false negatives through simplification to bytecode, and does not impose unwanted restrictions on program development.

The translation strategy defined in Chapter 3 provides a technique to generate *SCJ-mSafe* programs from valid Level 1 SCJ input programs that conform to the specification outlined in [46]. The structured *SCJ-mSafe* language allows for the definition of a simpler checking technique that does not have to cater for all of the complex commands and expressions that are possible in Java. It also provides the foundations to formally prove that the technique is sound, as discussed in the conclusions from Chapter 4.

The checking technique defined in Chapter 4 provides a method to check *SCJ-*

mSafe programs for possible memory-safety violations. One of the most interesting aspects of the technique is the generation of individual method properties, which provide a summary of a method's behaviour independently of its execution. The technique also uses an environment to capture the necessary information about aliasing and reference contexts of objects to facilitate the detection of possible memory-safety violations. The environment uses a flat structure to represent all expressions of an *SCJ-mSafe* program and is built up and maintained throughout the analysis.

A set of memory-safety inference rules have been defined that state what it means for a particular *SCJ-mSafe* component to be memory safe. Using the method properties, environment, and these rules, it is possible to establish the precise point in a program where a possible memory-safety violation may occur. As the technique uses static analysis, the compromise to the approach is that a worst-case view of the program must be taken, which means false-negatives may be raised during the analysis.

Finally, in the previous chapter, a number of test examples and case studies have been applied to the *TransMSafe* tool, which implements the translation and checking techniques. The results of the examples demonstrate the technique's ability to detect memory-safety violations in SCJ programs. The implementation of the tools also verify that it is possible to perform such checks automatically.

The technique is able to find possible memory-safety violations without the need for additional user-added annotations. Moreover, this avoids the restrictions that are imposed on SCJ programs by the addition of annotations; if an input SCJ program is valid according to the specification, it can be checked with the technique presented here, unlike the SCJChecker, which may require code duplication to handle some SCJ programs.

The bytecode analysis technique described has similar results to that presented here. It is faster at performing the analysis of a program, however, there are certain SCJ-specific methods that have not been handled, but this is a simple addition.

6.2 Conclusions

The main contribution of this thesis is a new static checking technique that is able to detect memory-safety violations using source-code analysis for the SCJ programming paradigm and memory model. The entire technique is formalised in Z to precisely define the languages, translation, and checking strategies involved; this formalisation can be used as the starting point to prove the soundness of the technique. For example, it is possible to prove the following.

$$mSafe(Translate(p)) \Rightarrow memorySafe(model[[p]])$$

The above states that, given an SCJ program p, the proof that mSafe holds, for the translation of p characterised by the *Translate* function defined in Chapter 3, using the rules for mSafe defined in Chapter 4 is possible only if the program p can be characterised as memory safe using its semantics defined by a semantic function *model* for SCJ.. Proving this would prove the soundness of the technique, that is, given a memory-safety violation in an SCJ program, the representation and analysis of that program in *SCJ-mSafe* will also identify the error. The above is not true in reverse; more specifically, it is not true that $mSafe(Translate(p))$ will always result in a memory-safe verdict for p even if the semantic definition states that p is error free. This is because the technique is sound, but not complete, and it may raise false negative, which is a sacrifice made when performing worst-case static analysis.

Other techniques such as the bytecode analysis technique in [14] perform similarly to the approach defined here in the sense that neither require annotations to detect possible memory-safety violations, however, there are differences between the two techniques. The intermediate representation of the source code used here is *SCJ-mSafe*; the representation of the source code used in the bytecode checking technique is Java bytecode. The difference between analysis of *SCJ-mSafe* and analysis of bytecode is reflected in issues of traceability and execution tracking. More specifically, by using *SCJ-mSafe*, which is an abstraction of the original source code as opposed to a transformation (like bytecode), it is much easier to map the points at which potential memory-safety errors are discovered back to the precise location in the source code. Secondly, the structure of the program and commands is maintained in *SCJ-mSafe*, which facilitates a well-defined flow of analysis throughout the program. It is easy to identify when missions are started, for example. At the bytecode level, this is not as easy, and as such the bytecode analysis technique encounters false negatives as a result of the simplifications made during compilation.

The method properties used in the technique have demonstrated the ability to perform modular reasoning and pre-analysis error detection for the SCJ memory model using an assertion (or postcondition) based technique. Whilst it is not always possible to determine the precise result for all reference variables or fields in advance of the analysis, there are situations where it is possible to detect errors that will always occur regardless of the execution context. A modular reasoning approach also reduces analysis time for largescale programs.

The memory-safety inference rules defined for *SCJ-mSafe* are clear and concise, which is made possible by the simple commands and expressions used in the *SCJ-mSafe* language. The simplification of complex SCJ statements during the translation from SCJ to *SCJ-mSafe* has lead to fewer rules that are easy to understand. This is not only an advantage to the analysis of SCJ programs, but to the overall new programming paradigm of safety-critical systems on which the rules are based: a well-defined programming paradigm that uses simple commands and expressions leads to simple and comprehensible rules for memory safety.

The *TransMSafe* tool has both demonstrated and validated that the formalised technique is, in fact, applicable to SCJ programs. The results of the examples and case studies in the previous chapter illustrate the power of the technique to not only pick up well-defined and known errors, but also unknown errors in larger examples that are difficult to analyse by hand. The time taken to analyse these programs is also pleasing, as the algorithms used in the implementation of the tool are not as efficient as they could be. The application of the tool to very-large and complex systems is yet to be seen as such examples in SCJ do not exist yet; the results from the previous chapter are, however, positive.

6.3 Further work

The technique presented here could be extended to facilitate analysis of a larger set of SCJ programs and optimise the performance of the tool. A proof of soundness would give additional confidence in the approach; a number of ideas for further work are described below.

6.3.1 Level 0 programs

Currently the technique is only applicable to Level 1 SCJ programs. Level 0 programs also suffer from possible memory-safety violations as they have the same hierarchy of memory areas. The main difference between handling Level 0 and Level 1 programs is the structure of the input SCJ program. One possible piece of further work is to extend the translation strategy to accommodate the structure of Level 0 programs, and translate them into the existing *SCJ-mSafe* structure for analysis.

6.3.2 Level 2 programs

Similarly, Level 2 programs are not supported by the technique. The translation strategy for Level 1 programs is applicable to Level 2 SCJ programs, however the checking technique is not applicable. In order to extend the checking technique to handle Level 2 programs, it would be necessary to enrich the definition of a reference context to handle the analysis of concurrent and nested missions; it is not sufficient to simply define the mission memory area, as multiple missions may be executing simultaneously. One possible method to overcome this would be to introduce an additional parameter on the mission reference contexts that identifies the expression associated with the particular mission object. This mission expression would also be need to be associated with any associated handler per-release reference context. Finally, temporary-private reference contexts that are currently associated with a handler name and nesting level would need to be associated with the mission expression, handler, and the nesting level.

6.3.3 Mutual recursion

Currently the technique is capable of handling simple method recursion and does not cater for mutual recursion. This is because no SCJ example discovered has included mutual recursion, however support for mutual recursion is a possibility in the technique. As recursive method properties are generated with a single pass of the method body, mutual recursion can also be handled by first analysing the bodies of each method whilst ignoring method calls to the dependant method. Having generated partial method properties for each, the full method properties for each can be calculated by re-running the analysing and taking into account the partial method properties of the dependant method.

6.3.4 Soundness

Having defined a formal model of the technique, it should be possible to prove the soundness of the technique. As mentioned previously, a formal model of the memory-model of SCJ has been defined in the UTP [11]; such a model would be a good starting point to prove the soundness of the approach.

6.3.5 Optimisation

As mentioned in the previous chapter, the implementation of the *TransMSafe* tool could be optimised to improve translation and analysis times, this is because the initial implementation was undertaken as a proof-of-concept. A further optimisation would be to investigate the implications of not using the reflexive, symmetric, transitive closure of the expression share relation, but instead keeping a simpler representation of which objects are aliased.

6.3.6 A more precise environment

The environment used to store the information required to determine possible memory-safety violations could be enriched to store more precise information about the analysis of a program. For example, the current model is a pair that captures the aliasing and references information for the whole program. It is possible to use a function instead that maps aliasing information to associated reference information. This would allow more precise versions of the current aliasing of a program to be stored simultaneously, subsequently reducing the number of false-negatives raised during analysis. A more precise aliasing would be possible when analysing conditional statements, for example. If the true branch produces one possible aliasing whilst the false branch produces another, the resulting environment could store both possibilities along with the associated reference information for each.

6.3.7 Automatic SCJ annotation

As described in the previous chapter, the generation of method properties for an *SCJ-mSafe* program can be considered as the automatic annotation of every method with a set of method properties. Another piece of possible further work would be to map these automatically generated method properties for *SCJ-mSafe* programs back to the corresponding methods in SCJ.

6.3.8 Application to other languages

The technique presented here is based on the new programming paradigm of SCJ and the scoped-based memory model that complements it. The fact that Java has been the language used to implement the new paradigm is potentially irrelevant, and the checking technique defined here could be applied to other languages that adopt a similar paradigm or memory model. Further work could investigate the application of this technique to other languages, or even investigate the application of the paradigm itself to other languages.

Appendix A

Z notation

Z notation	**Description**
$\pfun$	Partial function
$\leftrightarrow$	Relation
$\forall$	Universal quantification
$\exists$	Existential quantification
$\neg$	Not
$\times$	Cross
$\wedge$	Conjunction
$\vee$	Disjunction
$\mapsto$	Maps to
seq	Sequence
$\langle\rangle$	Empty sequence
$\frown$	Sequencer concatenation
head	Head of a sequence
tail	Tail of a sequence
$\mathbb{P}$	Power set
$\mathbb{F}$	Finite set
$\varnothing$	Empty set
$\cup$	Set union
$\setminus$	Set difference
$\in$	Set membership
$\subset$	Subset
$\subseteq$	Subset or equal to
$\oplus$	Override
*	Reflexive transitive closure
$^\sim$	Inverse
dom	Domain
ran	Range

Appendix B

SCJ model in Z

[*ReturnType*, *Body*, *Annotation*]

```
── TypeParameter ──────────────────────
│ name: Name
└──────────────────────────────────────
```

```
── TypeElement ────────────────────────
│ name: Name
│ typeParameters: seq TypeParameter
└──────────────────────────────────────
```

[*Value*]

```
│ safelet, missionSequencer, mission, APeriodicHandler, PeriodicHandler: Name
```

```
│ handleEvent, initialize, cleanUp, getSequencer, initializeApplication,
│ getNextMission: Name
```

```
│ void, run, Result, Object, Unknown, Empty: Name
```

```
│ executeInAreaOfID, executeInOuterAreaID, getMemoryAreaID, newArrayID,
│ newInstanceID, enterPrivateMemoryID, PeriodicEventHandler,
│ AperiodicEventHandler, register: Name
```

$$
\begin{array}{ll}
Flag ::= & abstract \\
 | & final \\
 | & native \\
 | & private \\
 | & protected \\
 | & public \\
 | & static \\
 | & strictfp \\
 | & synchronized \\
 | & transient \\
 | & volatile
\end{array}
$$

$$
\begin{array}{ll}
SCJExpression ::= & null \\
 | & annotation \\
 | & arrayAccess \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
 | & assignment \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
 | & binary \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
 | & compoundAssignment \langle\langle SCJExpression \times SCJExpression \rangle\rangle \\
 | & conditional \langle\langle SCJExpression \times SCJExpression \times SCJExpression \rangle\rangle \\
 | & erroneous \\
 | & identifier \langle\langle Name \rangle\rangle \\
 | & instanceOf \langle\langle SCJExpression \times TypeElement \rangle\rangle \\
 | & literal \langle\langle Value \rangle\rangle \\
 | & memberSelect \langle\langle SCJExpression \times Name \rangle\rangle \\
 | & methodInvocation \langle\langle SCJExpression \times \mathrm{seq}\ SCJExpression \rangle\rangle \\
 | & newArray \langle\langle TypeElement \times \mathrm{seq}\ SCJExpression \rangle\rangle
\end{array}
$$

$\quad \mid newClass\ \langle\langle Name \times \mathrm{seq}\ SCJExpression \rangle\rangle$
$\quad \mid parenthesized\ \langle\langle SCJExpression \rangle\rangle$
$\quad \mid typeCast\ \langle\langle TypeElement \times SCJExpression \rangle\rangle$
$\quad \mid unary\ \langle\langle SCJExpression \rangle\rangle$

$WellTypedExprs : \mathbb{P}\ SCJExpression$

$WellTypedExprs \subset SCJExpression$

SCJModifier

$flags : \mathbb{P}\ Flag$
$annotations : \mathbb{P}\ Annotation$

SCJVariable

$mods : SCJModifier$
$type : TypeElement$
$name : Name$
$init : SCJExpression$

$SCJCommand ::= assert\ \langle\langle SCJExpression \times SCJExpression \rangle\rangle$
$\quad \mid block\ \langle\langle Boolean \times SCJCommand \rangle\rangle$
$\quad \mid break\ \langle\langle Name \rangle\rangle$
$\quad \mid continue\ \langle\langle Name \rangle\rangle$
$\quad \mid doWhile\ \langle\langle SCJExpression \times SCJCommand \rangle\rangle$
$\quad \mid empty$
$\quad \mid eFor\ \langle\langle SCJModifier \times TypeElement \times Name \times SCJCommand \times$
$\qquad SCJExpression \times SCJCommand \rangle\rangle$
$\quad \mid expression\ \langle\langle SCJExpression \rangle\rangle$
$\quad \mid for\ \langle\langle SCJCommand \times SCJExpression \times SCJCommand \times SCJCommand \rangle\rangle$
$\quad \mid if\ \langle\langle SCJExpression \times SCJCommand \times SCJCommand \rangle\rangle$
$\quad \mid labeled\ \langle\langle Name \times SCJCommand \rangle\rangle$
$\quad \mid return\ \langle\langle SCJExpression \rangle\rangle$
$\quad \mid switch\ \langle\langle SCJModifier \times TypeElement \times Name \times SCJCommand \times$
$\qquad SCJExpression \times \mathrm{seq}\ SCJCommand \rangle\rangle$
$\quad \mid synchronized1\ \langle\langle SCJExpression \times Boolean \times SCJCommand \rangle\rangle$
$\quad \mid throw\ \langle\langle SCJExpression \rangle\rangle$
$\quad \mid try\ \langle\langle SCJCommand \times \mathrm{seq}\ SCJExpression \times \mathrm{seq}\ SCJCommand \times$
$\qquad SCJCommand \rangle\rangle$
$\quad \mid variable\ \langle\langle SCJVariable \rangle\rangle$
$\quad \mid while\ \langle\langle SCJExpression \times SCJCommand \rangle\rangle$

$WellTypedComs : \mathbb{P}\ SCJCommand$

$WellTypedComs \subset SCJCommand$

SCJMethod

$modifiers : SCJModifier$
$typeParameters : \mathbb{P}\ TypeParameter$
$returnType : TypeElement$
$name : Name$

$params: \mathrm{seq}\ SCJVariable$
$body: \mathrm{seq}\ SCJCommand$

$SCJClassComponent ::= ClassField\ \langle\langle SCJVariable \rangle\rangle \mid ClassMethod\ \langle\langle SCJMethod \rangle\rangle$

SCJClass

$modifiers: SCJModifier$
$name: Name$
$typeParameters: \mathbb{P}\ TypeParameter$
$extends: Name$
$implements: Name$
$members: \mathrm{seq}\ SCJClassComponent$

SCJProgram

$classes: \mathbb{P}\ SCJClass$

Appendix C

SCJ-mSafe model in Z

$Boolean ::= True \mid False$

$[Name]$

$Type ::= Primitive \mid Ref \langle\langle Name \rangle\rangle \mid Array \langle\langle Type \rangle\rangle$

VarType

$type: Name$
$isArray: Boolean$
$isPrimitive: Boolean$
$isReference: Boolean$
$resultVar: Boolean$

$isPrimitive \neq isReference$

Variable

$name: Name$
$varType: VarType$

ArrayElement

$name: Name$
$type: Name$

$Identifier ::= var \langle\langle Variable \rangle\rangle \mid arrayElement \langle\langle ArrayElement \rangle\rangle$

$seqtwo[X] == \{ s: \mathrm{seq}\, X \mid \# s > 1 \}$

$FieldAccess == seqtwo[Identifier]$

$Expr ::= Val \mid ID \langle\langle Identifier \rangle\rangle \mid FA \langle\langle FieldAccess \rangle\rangle \mid OtherExpr \mid Null \mid This$

$LExpr == \mathrm{ran}\, ID \cup \mathrm{ran}\, FA$

Dec

$var: Variable$

$RefCon ::= Prim \mid IMem \mid MMem \mid PRMem \langle\langle Name \rangle\rangle \mid TPMem \langle\langle Name \times \mathbb{N} \rangle\rangle \mid TPMMem \langle\langle \mathbb{N} \rangle\rangle$

$MetaRefCon ::= Rcs \langle\langle \mathbb{P}\, RefCon \rangle\rangle$
$\mid Erc \langle\langle LExpr \rangle\rangle$
$\mid Current$
$\mid CurrentPrivate \langle\langle \mathbb{N} \rangle\rangle$
$\mid CurrentPlus \langle\langle \mathbb{N} \rangle\rangle$

MethodSig

$name: Name$
$class: Name$
$classExtends: Name$

descendants: $\mathbb{P}$ *Name*
returnType: Type
returnTypeName: Name
paramTypes: seq *Name*

methodCall

le: LExpr
name: Name
args: seq *Expr*
methods: $\mathbb{P}$ *MethodSig*

newInstance

le: LExpr
mrc: MetaRefCon
type: VarType
args: seq *Expr*

getMemoryArea

ref: LExpr
e: LExpr

Com ::= *Skip*
| *Decl* ⟪*Dec*⟫
| *Asgn* ⟪*LExpr* × *Expr*⟫
| *Seq* ⟪*Com* × *Com*⟫
| *Scope* ⟪*Com*⟫
| *NewInstance* ⟪*newInstance*⟫
| *If* ⟪*Expr* × *Com* × *Com*⟫
| *Switch* ⟪*Expr* × seq *Com*⟫
| *For* ⟪*Com* × *Expr* × *Com* × *Com*⟫
| *MethodCall* ⟪*methodCall*⟫
| *EnterPrivateMemory* ⟪*methodCall*⟫
| *ExecuteInAreaOf* ⟪*MetaRefCon* × *methodCall*⟫
| *ExecuteInOuterArea* ⟪*methodCall*⟫
| *GetMemoryArea* ⟪*getMemoryArea*⟫
| *Try* ⟪*Com* × seq *Expr* × seq *Com* × *Com*⟫
| *While* ⟪*Expr* × *Com*⟫
| *DoWhile* ⟪*Com* × *Expr*⟫

Method

name: Name
returnType: Name
type: Type
params: seq *Variable*
class: Name
properties: MethodProperties
body: Com
localVars: $\mathbb{P}$ *LExpr*

visibleFields: $\mathbb{P}$ *LExpr*

Class

name: *Name*
extends: *Name*
embeddedIn: *Name*
fields: seq *Dec*
init: seq *Com*
constrs: $\mathbb{P}$ *Method*
methods: $\mathbb{P}$ *Method*

Handler

name: *Name*
fields: seq *Dec*
init: seq *Com*
constrs: $\mathbb{P}$ *Method*
methods: $\mathbb{P}$ *Method*
hAe: *Com*

Mission

name: *Name*
fields: seq *Dec*
init: seq *Com*
constrs: $\mathbb{P}$ *Method*
methods: $\mathbb{P}$ *Method*
initialize: *Com*
handlers: $\mathbb{P}$ *Name*
cleanUp: *Com*

MissionSeq

name: *Name*
fields: seq *Dec*
init: seq *Com*
constrs: $\mathbb{P}$ *Method*
methods: $\mathbb{P}$ *Method*
missions: $\mathbb{P}$ *Name*
getNextMission: *Com*

Safelet

name: *Name*
fields: seq *Dec*
init: seq *Com*
constrs: $\mathbb{P}$ *Method*
methods: $\mathbb{P}$ *Method*
initializeApplication: *Com*
getSequencer: *Com*

missionSeq: MissionSeq

SCJmSafeProgram

static: $\mathbb{P}$ *Dec*
sInit: $\mathbb{P}$ *Com*
safelet: *Safelet*
missionSeq: *MissionSeq*
missions: $\mathbb{P}$ *Mission*
handlers: $\mathbb{P}$ *Handler*
classes: $\mathbb{P}$ *Class*

Appendix D

Translation strategy in Z

```
│ WellTypedProgs: ℙ SCJProgram
├──────────────────────────
│ WellTypedProgs ⊂ SCJProgram
```

```
┌── TranslationEnv ──────────────────────────────
│ variables: ℙ Variable
│ methods: ℙ MethodSig
└────────────────────────────────────────────────
```

```
│ ExtractExpression: SCJExpression ⇸ Expr
├──────────────────────────
│ dom ExtractExpression ⊂ WellTypedExprs
│ ∧ (∀ scjExpr: dom ExtractExpression
│     • (scjExpr = null ∧ ExtractExpression scjExpr = Null
│       ∨ scjExpr = annotation ∧ ExtractExpression scjExpr = OtherExpr
│       ∨ (∃ e1, e2: SCJExpression; iden: Identifier
│           | scjExpr = arrayAccess (e1, e2)
│           • (let arrayExpr == ExtractExpression e1
│               • ((∃ iden: Identifier; v: Variable; ae: ArrayElement
│                     | arrayExpr = ID iden
│                       ∧ iden = var v
│                       ∧ ae.name = v.name
│                     • ExtractExpression scjExpr
│                       = ID (arrayElement ae))
│                  ∨ (∃ iden: Identifier; ae: ArrayElement
│                       | arrayExpr = ID iden ∧ iden = arrayElement ae
│                       • ExtractExpression scjExpr = ID iden)
│                  ∨ (∃ fa: FieldAccess; v: Variable; ae: ArrayElement
│                       | arrayExpr = FA fa ∧ v.name = ae.name
│                       • last fa = var v
│                         ∧ ExtractExpression scjExpr
│                           = FA (front fa ⁀ ⟨arrayElement ae⟩))))))
│       ∨ (∃ e1, e2: SCJExpression | scjExpr = assignment (e1, e2)
│           • ExtractExpression scjExpr = ExtractExpression e1)
│       ∨ (∃ e1, e2: SCJExpression | scjExpr = binary (e1, e2)
│           • ExtractExpression scjExpr = Val)
│       ∨ (∃ e1, e2: SCJExpression | scjExpr = compoundAssignment (e1, e2)
│           • ExtractExpression scjExpr = ExtractExpression e1)
│       ∨ (∃ e1, e2, e3: SCJExpression | scjExpr = conditional (e1, e2, e3)
│           • ExtractExpression scjExpr = OtherExpr)
│       ∨ scjExpr = erroneous ∧ ExtractExpression scjExpr = OtherExpr
│       ∨ (∃ name: Name; iden: Identifier; v: Variable
│           | scjExpr = identifier name ∧ iden = var v ∧ v.name = name
│           • ExtractExpression scjExpr = ID iden)
│       ∨ (∃ e1: SCJExpression; type: TypeElement
│           | scjExpr = instanceOf (e1, type)
│           • ExtractExpression scjExpr = OtherExpr)
│       ∨ (∃ value: Value | scjExpr = literal value
│           • ExtractExpression scjExpr = Val)
│       ∨ (∃ e1: SCJExpression; name: Name; fa, fa2: FieldAccess;
│           iden: Identifier; v: Variable
│           | scjExpr = memberSelect (e1, name) ∧ v.name = name
│           • (let lhs == ExtractExpression e1
│               • (lhs = ID iden
                    $\land fa = \langle iden \rangle ^\frown \langle var\ v \rangle$
                    $\land ExtractExpression\ scjExpr = FA\ fa$
                    $\lor lhs = FA\ fa2$
                      $\land fa = fa2 ^\frown \langle var\ v \rangle$
                      $\land ExtractExpression\ scjExpr = FA\ fa$
                    $\lor lhs \neq ID\ iden$
                      $\land lhs \neq FA\ fa2$
                      $\land ExtractExpression\ scjExpr = OtherExpr)))$
        $\lor (\exists\ identifier: SCJExpression; e1: \mathrm{seq}\ SCJExpression$
              $\mid scjExpr = methodInvocation\ (identifier, e1)$
              $\bullet ExtractExpression\ scjExpr = OtherExpr)$
        $\lor (\exists\ type: TypeElement; args: \mathrm{seq}\ SCJExpression$
              $\mid scjExpr = newArray\ (type, args)$
              $\bullet ExtractExpression\ scjExpr = OtherExpr)$
        $\lor (\exists\ name: Name; args: \mathrm{seq}\ SCJExpression$
              $\mid scjExpr = newClass\ (name, args)$
              $\bullet ExtractExpression\ scjExpr = OtherExpr)$
        $\lor (\exists\ e1: SCJExpression \mid scjExpr = parenthesized\ e1$
              $\bullet ExtractExpression\ scjExpr = ExtractExpression\ e1)$
        $\lor (\exists\ type: TypeElement; e1: SCJExpression$
              $\mid scjExpr = typeCast\ (type, e1)$
              $\bullet ExtractExpression\ scjExpr = ExtractExpression\ e1)$
        $\lor (\exists\ e1: SCJExpression \mid scjExpr = unary\ e1$
              $\bullet ExtractExpression\ scjExpr = ExtractExpression\ e1)))$
```

```
$SimplifyCommandPair: Com \times Com \rightarrow\!\!\!\!\!\!+\ Com$
──────────────
$\forall\ c1, c2: Com$
  $\bullet\ c1 = Skip \land \neg\ c2 = Skip \land SimplifyCommandPair\ (c1, c2) = c2$
    $\lor \neg\ c1 = Skip \land c2 = Skip \land SimplifyCommandPair\ (c1, c2) = c1$
    $\lor \neg\ c1 = Skip$
      $\land \neg\ c2 = Skip$
      $\land SimplifyCommandPair\ (c1, c2) = Seq\ (c1, c2)$
```

```
$GetMethodCallReturnDec: methodCall \times Com \rightarrow\!\!\!\!\!\!+\ Expr \times Com$
──────────────
$\forall\ c1: methodCall; c2: Com$
  $\bullet\ \exists\ mc: methodCall; v1, v2: Variable; n: Name; d: Dec$
    $\bullet\ (\#\ mc.args = 0$
      $\lor \neg\ last\ mc.args = ID\ (var\ v2)$
      $\lor last\ mc.args = ID\ (var\ v2) \land v2.varType.resultVar = False)$
      $\land c1 = mc$
      $\land v1.name = n$
      $\land v1.varType.type = Unknown$
      $\land d.var = v1$
      $\land GetMethodCallReturnDec\ (c1, c2)$
         $= (ID\ (var\ v1), Seq\ ((Decl\ d), c2))$
      $\lor \#\ mc.args > 0$
        $\land last\ mc.args = ID\ (var\ v2)$
        $\land v2.varType.resultVar = True$
        $\land GetMethodCallReturnDec\ (c1, c2) = (ID\ (var\ v2), c2)$
```

```
$CreateSingleCommand: \mathrm{seq}\ Com \rightarrow\!\!\!\!\!\!+\ Com$
```

```
∀ seqCom: seq Com
  • # seqCom > 1
    ∧ CreateSingleCommand seqCom
        = Seq ((head seqCom), (CreateSingleCommand (tail seqCom)))
    ∨ # seqCom = 1 ∧ CreateSingleCommand seqCom = head seqCom
    ∨ # seqCom = 0 ∧ CreateSingleCommand seqCom = Skip
```

```
MergeSideEffectsParamsCom: Com × seq Com × Com ⇸ Com
────────────────────────────
∀ c1, c2, c3: Com; seqCom: seq Com | c3 = CreateSingleCommand seqCom
  • c2 = Skip
    ∧ (# seqCom = 0 ∧ MergeSideEffectsParamsCom (c1, seqCom, c2) = c1
      ∨ (∃ c4, c5: Com
            • # seqCom > 0
              ∧ (c1 = Seq (c4, c5)
                 ∧ MergeSideEffectsParamsCom (c1, seqCom, c2)
                     = Seq ((Seq (c3, c4)), c5)
                 ∨ ¬ c1 = Seq (c4, c5)
                   ∧ MergeSideEffectsParamsCom (c1, seqCom, c2)
                       = Seq (c3, c1))))
    ∨ ¬ c2 = Skip
      ∧ (∃ c4, c5: Com
            • (# seqCom = 0
              ∧ (c1 = Seq (c4, c5)
                 ∧ MergeSideEffectsParamsCom (c1, seqCom, c2)
                     = Seq ((Seq (c2, c4)), c5)
                 ∨ ¬ c1 = Seq (c4, c5)
                   ∧ MergeSideEffectsParamsCom (c1, seqCom, c2)
                       = Seq (c2, c1))
              ∨ # seqCom > 0
                ∧ c1 = Seq (c4, c5)
                ∧ MergeSideEffectsParamsCom (c1, seqCom, c2)
                    = Seq ((Seq ((Seq (c2, c3)), c4)), c5)
              ∨ ¬ c1 = Seq (c4, c5)
                ∧ MergeSideEffectsParamsCom (c1, seqCom, c2)
                    = Seq ((Seq (c2, c3)), c1)))
```

```
GetMethodName: SCJExpression ⇸ Name
────────────────────────────
∀ e: SCJExpression
  • (∃ n: Name | e = identifier n • GetMethodName e = n)
    ∨ (∃ e1: SCJExpression; n: Name | e = memberSelect (e1, n)
          • GetMethodName e = n)
```

```
ExtractExprType: Expr ⇸ Name
────────────────────────────
∀ e: Expr
  • (∃ fa: FieldAccess • ExtractExprType e = ExtractExprType (ID (last fa)))
    ∨ (∃ ae: ArrayElement • ExtractExprType e = ae.type)
    ∨ (∃ v: Variable • ExtractExprType e = v.varType.type)
```

$$
\begin{array}{|l}
\mathit{ExtractArgTypes}: \mathrm{seq}\ \mathit{Expr} ⇸ \mathrm{seq}\ \mathit{Name} \\
\hline
\forall\ e: \mathrm{seq}\ \mathit{Expr} \\
\quad \bullet\ \mathit{ExtractArgTypes}\ e \\
\qquad = \langle \mathit{ExtractExprType}\ (\mathit{head}\ e) \rangle ⁀ \mathit{ExtractArgTypes}\ (\mathit{tail}\ e)
\end{array}
$$

$$
\begin{array}{|l}
\mathit{FindMethods}: \mathit{Expr} \times \mathit{Name} \times \mathrm{seq}\ \mathit{Expr} \times \mathit{TranslationEnv} ⇸ \mathbb{P}\ \mathit{MethodSig} \\
\hline
\forall\ e: \mathit{Expr};\ n: \mathit{Name};\ \mathit{args}: \mathrm{seq}\ \mathit{Expr};\ \mathit{transEnv}: \mathit{TranslationEnv} \\
\quad \bullet\ \exists\ n1: \mathit{Name};\ \mathit{classes}: \mathbb{P}\ \mathit{Name};\ \mathit{argTypes}: \mathrm{seq}\ \mathit{Name} \\
\qquad \mid\ n1 = \mathit{ExtractExprType}\ e \\
\qquad\ \ \wedge\ \mathit{classes} \\
\qquad\qquad = \bigcup \{\ \mathit{ms}: \mathit{transEnv.methods} \mid \mathit{ms.class} = n1 \\
\qquad\qquad\qquad \bullet\ (\{\mathit{ms.class}\} \cup \mathit{ms.descendants})\ \} \\
\qquad\ \ \wedge\ \mathit{argTypes} = \mathit{ExtractArgTypes}\ \mathit{args} \\
\qquad \bullet\ \mathit{FindMethods}\ (e, n, \mathit{args}, \mathit{transEnv}) \\
\qquad\quad = \bigcup \{\ \mathit{ms}: \mathit{transEnv.methods} \\
\qquad\qquad\quad \mid\ \mathit{ms.class} \in \mathit{classes} \\
\qquad\qquad\quad\ \ \wedge\ \mathit{ms.name} = n \\
\qquad\qquad\quad\ \ \wedge\ \#\ \mathit{ms.paramTypes} = \#\ \mathit{argTypes} \\
\qquad\qquad\quad \bullet\ \{\ n1: \mathit{ms.paramTypes};\ n2: \mathit{argTypes} \mid n1 = n2 \bullet \mathit{ms}\ \}\ \}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{FindMethodType}: \mathit{Expr} \times \mathit{Name} \times \mathrm{seq}\ \mathit{Expr} \times \mathit{TranslationEnv} ⇸ \mathit{Name} \\
\hline
\forall\ e: \mathit{Expr};\ n: \mathit{Name};\ \mathit{args}: \mathrm{seq}\ \mathit{Expr};\ \mathit{transEnv}: \mathit{TranslationEnv} \\
\quad \bullet\ \exists\ \mathit{methods}: \mathbb{P}\ \mathit{MethodSig} \mid \mathit{methods} = \mathit{FindMethods}\ (e, n, \mathit{args}, \mathit{transEnv}) \\
\qquad \bullet\ \#\ \mathit{methods} = 0 \wedge \mathit{FindMethodType}\ (e, n, \mathit{args}, \mathit{transEnv}) = \mathit{void} \\
\qquad\ \ \vee\ (\exists\ \mathit{ms}: \mathit{methods} \\
\qquad\qquad\quad \bullet\ \#\ \mathit{methods} = 1 \\
\qquad\qquad\quad\ \ \wedge\ \mathit{FindMethodType}\ (e, n, \mathit{args}, \mathit{transEnv}) \\
\qquad\qquad\qquad = \mathit{ms.returnTypeName}) \\
\qquad\ \ \vee\ \#\ \mathit{methods} > 1 \\
\qquad\quad \wedge\ (\exists\ \mathit{types}: \mathbb{P}\ \mathit{Name} \\
\qquad\qquad\quad \mid\ \mathit{types} = \{\ \mathit{ms}: \mathit{methods} \bullet \mathit{ms.returnTypeName}\ \} \\
\qquad\qquad\quad \bullet\ ((\exists\ t: \mathit{Name} \\
\qquad\qquad\qquad\quad \bullet\ t \in \mathit{types} \\
\qquad\qquad\qquad\qquad \wedge\ (\forall\ t2: \mathit{Name} \mid t2 \notin \mathit{types} \wedge t \neq t2 \\
\qquad\qquad\qquad\qquad\quad \bullet\ \mathit{FindMethodType}\ (e, n, \mathit{args}, \mathit{transEnv}) \\
\qquad\qquad\qquad\qquad\qquad = t)) \\
\qquad\qquad\qquad \vee\ (\exists\ t, t2: \mathit{Name} \mid t \in \mathit{types} \wedge t2 \in \mathit{types} \wedge t \neq t2 \\
\qquad\qquad\qquad\qquad \bullet\ \mathit{FindMethodType}\ (e, n, \mathit{args}, \mathit{transEnv}) \\
\qquad\qquad\qquad\qquad\quad = \mathit{void})))
\end{array}
$$

$$
\begin{array}{|l}
\mathit{TranslateExpression}: \mathit{SCJExpression} \times \mathit{TranslationEnv} ⇸ \mathit{Com} \\
\hline
\quad \forall\ \mathit{scjExpr}: \mathit{SCJExpression};\ \mathit{transEnv}: \mathit{TranslationEnv} \\
\quad \mid (\mathit{scjExpr}, \mathit{transEnv}) \in \mathrm{dom}\ \mathit{TranslateExpression} \\
\qquad \wedge\ \mathit{scjExpr} \in \mathit{WellTypedExprs} \\
\quad \bullet\ \mathit{scjExpr} = \mathit{null} \wedge \mathit{TranslateExpression}\ (\mathit{scjExpr}, \mathit{transEnv}) = \mathit{Skip} \\
\qquad \vee\ \mathit{scjExpr} = \mathit{annotation} \wedge \mathit{TranslateExpression}\ (\mathit{scjExpr}, \mathit{transEnv}) = \mathit{Skip} \\
\qquad \vee\ (\exists\ e1, e2: \mathit{SCJExpression} \mid \mathit{scjExpr} = \mathit{arrayAccess}\ (e1, e2) \\
\qquad\qquad \bullet\ \mathit{TranslateExpression}\ (\mathit{scjExpr}, \mathit{transEnv}) \\
\qquad\qquad\quad = \mathit{SimplifyCommandPair}\ ((\mathit{TranslateExpression}\ (e1, \mathit{transEnv})),
\end{array}
$$

```
                              (TranslateExpression (e2,
                                                    transEnv))))
   ∨ (∃ e1, e2: SCJExpression; lexpr, rexpr: Expr
        | scjExpr = assignment (e1, e2)
         ∧ lexpr = ExtractExpression e1
         ∧ rexpr = ExtractExpression e2
      • ((∃ type: TypeElement; args: seq SCJExpression;
            nI: newInstance
             | e2 = newArray (type, args)
              ∧ nI.le = lexpr
              ∧ nI.mrc = Current
              ∧ nI.type.type = type.name
              ∧ nI.type.isReference = True
              ∧ nI.type.isArray = True
              ∧ nI.args
                 = {  n: 1 .. # args
                      • n ↦ ExtractExpression (args n) }
           • TranslateExpression (scjExpr, transEnv)
              = MergeSideEffectsParamsCom ((NewInstance nI),
                                     {  n: 1 .. # args
                                        • (n
                                          ↦ TranslateExpression ((args n),
                                                               transEnv)) },
                                     Skip))
         ∨ (∃ args: seq SCJExpression; nI: newInstance; name: Name
              | e2 = newClass (name, args)
               ∧ nI.type.type = name
               ∧ nI.type.isReference = True
               ∧ nI.le = lexpr
               ∧ nI.mrc = Current
               ∧ nI.args
                  = {  n: 1 .. # args
                       • n ↦ ExtractExpression (args n) }
            • TranslateExpression (scjExpr, transEnv)
               = MergeSideEffectsParamsCom ((NewInstance nI),
                                      {  n: 1 .. # args
                                         • (n
                                           ↦ TranslateExpression ((args n),
                                                                transEnv)) },
                                      Skip))
         ∨ (∃ le: SCJExpression; args: seq SCJExpression; c: Com
              | e2 = methodInvocation (le, args)
               ∧ c = TranslateExpression (e2, transEnv)
            • ((∃ mc: methodCall | c = MethodCall mc
                 • TranslateExpression (scjExpr, transEnv)
                    = Seq ((GetMethodCallReturnDec (mc, c)).2,
                       (Asgn (lexpr,
                            (GetMethodCallReturnDec (mc,
                                                    c)).1))))
               ∨ (∃ c1, c2: Com | c = Seq (c1, c2)
                  • ((∃ mc: methodCall | c2 = MethodCall mc
                       • TranslateExpression (scjExpr,
                                              transEnv)
                          = Seq ((GetMethodCallReturnDec (mc,
                                                         c2)).2,
                             (Asgn (lexpr,
                                  (GetMethodCallReturnDec (mc,
                                                        c2)).1))))
                    ∨ (∃ nI: newInstance
                          | c2 = NewInstance nI
                            ∧ nI.le = lexpr
                          • TranslateExpression (scjExpr,
                                                transEnv)
                              = Seq (c1, (NewInstance nI)))
                    ∨ (∃ gMem1, gMem2: getMemoryArea
                          | c2 = GetMemoryArea gMem1
                            ∧ gMem2.e = gMem1.e
                            ∧ gMem2.ref = lexpr
                          • TranslateExpression (scjExpr,
                                                transEnv)
                              = Seq (c1,
                                    (GetMemoryArea gMem2)))))
                ∨ (∃ nI: newInstance
                      | c = NewInstance nI ∧ nI.le = lexpr
                      • TranslateExpression (scjExpr, transEnv)
                          = NewInstance nI)
                ∨ (∃ gMem: getMemoryArea
                      | c = GetMemoryArea gMem ∧ gMem.ref = lexpr
                      • TranslateExpression (scjExpr, transEnv)
                          = GetMemoryArea gMem)))
        ∨ (∃ e3, e4: SCJExpression | e2 = arrayAccess (e3, e4)
            • ((∃ e5: SCJExpression; e6: seq SCJExpression; c: Com
                    | e3 = methodInvocation (e5, e6)
                      ∧ c = TranslateExpression (e3, transEnv)
                    • (∃ mc: methodCall | c = MethodCall mc
                        • TranslateExpression (scjExpr,
                                              transEnv)
                            = SimplifyCommandPair ((SimplifyCommandPair (
                                                    (GetMethodCallReturnDec (mc,
                                                                          c)).2,
                                                  (TranslateExpression (e4,
                                                                    transEnv)))),
                                        (Asgn (lexpr,
                                            (GetMethodCallReturnDec (mc,
                                                                  c)).1))))
                    ∨ (∃ c1, c2: Com; mc: methodCall
                        | c = Seq (c1, c2)
                          ∧ c2 = MethodCall mc
                        • TranslateExpression (scjExpr,
                                              transEnv)
                            = SimplifyCommandPair ((SimplifyCommandPair (
                                                    (GetMethodCallReturnDec (mc,
                                                                          c)).2,
                                                  (TranslateExpression (e4,
                                                                    transEnv)))),
                                        (Asgn (lexpr,
                                            (GetMethodCallReturnDec (mc,
                                                                  c)).1)))))
                ∨ (∃ e5: SCJExpression; e6: seq SCJExpression;
                    c: Com
                      | e3 ≠ methodInvocation (e5, e6)
                        ∧ c = TranslateExpression (e3, transEnv)
                        • TranslateExpression (scjExpr, transEnv)
                            = SimplifyCommandPair ((SimplifyCommandPair (c,
                                                        (TranslateExpression (e4,
                                                                    transEnv)))),
                                            (Asgn (lexpr,
                                                rexpr))))))
            ∨ (∃ e3, e4: SCJExpression; args: seq SCJExpression;
                name: Name; type: TypeElement
                  | e2 ≠ newArray (type, args)
                    ∧ e2 ≠ newClass (name, args)
                    ∧ e2 ≠ methodInvocation (e3, args)
                    ∧ e2 ≠ arrayAccess (e3, e4)
                  • TranslateExpression (scjExpr, transEnv)
                      = SimplifyCommandPair ((TranslateExpression (e2,
                                                        transEnv)),
                                    (Asgn (lexpr, rexpr))))))
        ∨ (∃ e1, e2: SCJExpression; lcom, rcom, lcom2, rcom2: Com
            | scjExpr = binary (e1, e2)
              ∧ lcom = TranslateExpression (e1, transEnv)
              ∧ rcom = TranslateExpression (e2, transEnv)
            • ((∃ mc1, mc2: methodCall
                  • lcom = MethodCall mc1
                    ∧ ¬ rcom = MethodCall mc2
                    ∧ lcom2 = (GetMethodCallReturnDec (mc1, lcom)).2
                    ∧ rcom2 = rcom)
              ∨ (∃ mc1, mc2: methodCall
                  • rcom = MethodCall mc2
                    ∧ ¬ lcom = MethodCall mc1
                    ∧ rcom2 = (GetMethodCallReturnDec (mc2, rcom)).2
                    ∧ lcom2 = lcom)
              ∨ (∃ mc1, mc2: methodCall
                  • ¬ lcom = MethodCall mc1
                    ∧ ¬ rcom = MethodCall mc2
                    ∧ lcom2 = lcom
                    ∧ rcom2 = rcom))
              ∧ TranslateExpression (scjExpr, transEnv)
                  = SimplifyCommandPair (lcom2, rcom2))
        ∨ (∃ e1, e2: SCJExpression; sideEffect: Com
            | scjExpr = compoundAssignment (e1, e2)
              ∧ sideEffect = TranslateExpression (e2, transEnv)
            • (sideEffect = Skip
              ∧ TranslateExpression (scjExpr, transEnv)
                  = Asgn ((ExtractExpression e1), Val)
              ∨ ¬ sideEffect = Skip
                ∧ TranslateExpression (scjExpr, transEnv)
                    = Seq (sideEffect,
                          (Asgn ((ExtractExpression e1), Val)))))
        ∨ (∃ e1, e2, e3: SCJExpression | scjExpr = conditional (e1, e2, e3)
            • TranslateExpression (scjExpr, transEnv)
                = Seq ((TranslateExpression (e1, transEnv)),
                      (If ((ExtractExpression e1),
                          (TranslateExpression (e2, transEnv)),
                          (TranslateExpression (e3, transEnv))))))
        ∨ scjExpr = erroneous ∧ TranslateExpression (scjExpr, transEnv) = Skip
        ∨ (∃ name: Name | scjExpr = identifier name
            • TranslateExpression (scjExpr, transEnv) = Skip)
```

```
∨ (∃ e1: SCJExpression; type: TypeElement
     | scjExpr = instanceOf (e1, type)
     • TranslateExpression (scjExpr, transEnv)
         = TranslateExpression (e1, transEnv))
∨ (∃ value: Value | scjExpr = literal value
     • TranslateExpression (scjExpr, transEnv) = Skip)
∨ (∃ e1: SCJExpression; name: Name | scjExpr = memberSelect (e1, name)
     • TranslateExpression (scjExpr, transEnv)
         = TranslateExpression (e1, transEnv))
∨ (∃ e1: SCJExpression; args: seq SCJExpression; name, mname: Name; c,
   sideEffect: Com; lexpr: Expr; paramComs: seq Com;
   paramExprs: seq Expr
     | scjExpr = methodInvocation (e1, args)
       ∧ c = TranslateExpression (e1, transEnv)
       ∧ paramComs
           = {  n: 1 .. # args
                 • n ↦ TranslateExpression ((args n), transEnv) }
       ∧ paramExprs
           = {  n: 1 .. # args • n ↦ ExtractExpression (args n) }
       ∧ mname = GetMethodName e1
     • ((∃ c1, c2: Com; mc: methodCall
            | c = Seq (c1, c2) ∧ c2 = MethodCall mc
              ∨ c = MethodCall mc
            • lexpr = (GetMethodCallReturnDec (mc, c)).1
              ∧ sideEffect = (GetMethodCallReturnDec (mc, c)).2)
        ∨ (∃ c1, c2: Com; mc: methodCall
            | c ≠ MethodCall mc ∧ c ≠ Seq (c1, c2)
            • lexpr = ExtractExpression e1))
       ∧ (mname = executeInAreaOfID
          ∧ (∃ mc: methodCall
               | mc.methods
                   = FindMethods ((paramExprs 2), run, ⟨⟩,
                             transEnv)
                 ∧ mc.le = paramExprs 2
                 ∧ mc.args = ⟨⟩
                 ∧ mc.name = run
               • TranslateExpression (scjExpr, transEnv)
                   = MergeSideEffectsParamsCom ((ExecuteInAreaOf ((Erc (paramExprs 1)),
                                                       mc)),
                                     paramComs,
                                     sideEffect))
          ∨ mname = executeInOuterAreaID
            ∧ (∃ mc: methodCall
                 | mc.methods
                     = FindMethods ((paramExprs 1), run, ⟨⟩,
                               transEnv)
                   ∧ mc.le = paramExprs 1
                   ∧ mc.args = ⟨⟩
                   ∧ mc.name = run
                 • TranslateExpression (scjExpr, transEnv)
                     = MergeSideEffectsParamsCom ((ExecuteInOuterArea mc),
                                       paramComs,
                                       sideEffect))
          ∨ mname = getMemoryAreaID
            ∧ (∃ gMem: getMemoryArea
                 | gMem.ref = Null ∧ gMem.e = paramExprs 1
```

```
                • TranslateExpression (scjExpr, transEnv)
                    = MergeSideEffectsParamsCom ((GetMemoryArea gMem),
                                                  paramComs,
                                                  sideEffect))
        ∨ mname = newArrayID
          ∧ (∃ nI: newInstance; varType: VarType
                | nI.le = Null
                  ∧ nI.mrc = Erc lexpr
                  ∧ nI.type = varType
                  ∧ nI.args = ⟨⟩
                  ∧ varType.type = ExtractExprType (paramExprs 1)
                  ∧ varType.isReference = True
                  ∧ varType.isArray = True
                • TranslateExpression (scjExpr, transEnv)
                    = MergeSideEffectsParamsCom ((NewInstance nI),
                                                  paramComs,
                                                  sideEffect))
        ∨ mname = newInstanceID
          ∧ (∃ nI: newInstance; varType: VarType
                | nI.le = Null
                  ∧ nI.mrc = Erc lexpr
                  ∧ nI.type = varType
                  ∧ nI.args = ⟨⟩
                  ∧ varType.type = ExtractExprType (paramExprs 1)
                  ∧ varType.isReference = True
                • TranslateExpression (scjExpr, transEnv)
                    = MergeSideEffectsParamsCom ((NewInstance nI),
                                                  paramComs,
                                                  sideEffect))
        ∨ mname = enterPrivateMemoryID
          ∧ (∃ mc: methodCall
                | mc.methods
                    = FindMethods ((paramExprs 1), run, ⟨⟩,
                                   transEnv)
                  ∧ mc.le = paramExprs 1
                  ∧ mc.args = ⟨⟩
                  ∧ mc.name = run
                • TranslateExpression (scjExpr, transEnv)
                    = MergeSideEffectsParamsCom ((EnterPrivateMemory mc),
                                                  paramComs,
                                                  sideEffect))
        ∨ (∃ mc: methodCall; type: Name
              | mc.le = lexpr
                ∧ mc.name = mname
                ∧ mc.methods
                    = FindMethods (lexpr, mname, paramExprs,
                                   transEnv)
                ∧ type
                    = FindMethodType (lexpr, mname, paramExprs,
                                      transEnv)
              • (∃ d: Dec; v: Variable
                    | d.var = v
                      ∧ v.varType.type = type
                      ∧ v.varType.resultVar = True
                    • (type ≠ void
                       ∧ mc.args = paramExprs ⁀ ⟨ID (var v)⟩
```

```
                          ∧ TranslateExpression (scjExpr, transEnv)
                             = MergeSideEffectsParamsCom ((Seq ((Decl d),
                                                          (MethodCall mc))),
                                                       paramComs,
                                                       sideEffect)
                          ∨ type = void
                            ∧ mc.args = paramExprs
                            ∧ TranslateExpression (scjExpr,
                                                  transEnv)
                               = MergeSideEffectsParamsCom ((MethodCall mc),
                                                         paramComs,
                                                         sideEffect))))))
   ∨ (∃ type: TypeElement; args: seq SCJExpression; v: Variable; d: Dec;
      nI: newInstance
        | scjExpr = newArray (type, args)
          ∧ v.varType.type = type.name
          ∧ v.varType.isReference = True
          ∧ v.varType.isArray = True
          ∧ d.var = v
          ∧ nI.mrc = Current
          ∧ nI.le = ID (var v)
          ∧ nI.args
             = {  n: 1 .. # args • n ↦ ExtractExpression (args n) }
          ∧ nI.type = v.varType
        • TranslateExpression (scjExpr, transEnv)
           = MergeSideEffectsParamsCom ((NewInstance nI),
                                   {  n: 1 .. # args
                                       • (n
                                         ↦ TranslateExpression ((args n),
                                                                transEnv)) },
                                   (Decl d)))
   ∨ (∃ name: Name; args: seq SCJExpression; v: Variable; d: Dec;
      nI: newInstance
        | scjExpr = newClass (name, args)
          ∧ v.varType.type = name
          ∧ v.varType.isReference = True
          ∧ d.var = v
          ∧ nI.mrc = Current
          ∧ nI.le = ID (var v)
          ∧ nI.args
             = {  n: 1 .. # args • n ↦ ExtractExpression (args n) }
          ∧ nI.type = v.varType
        • TranslateExpression (scjExpr, transEnv)
           = MergeSideEffectsParamsCom ((NewInstance nI),
                                   {  n: 1 .. # args
                                       • (n
                                         ↦ TranslateExpression ((args n),
                                                                transEnv)) },
                                   (Decl d)))
   ∨ (∃ e1: SCJExpression | scjExpr = parenthesized e1
        • TranslateExpression (scjExpr, transEnv)
           = TranslateExpression (e1, transEnv))
   ∨ (∃ type: TypeElement; e1: SCJExpression
        | scjExpr = typeCast (type, e1)
        • TranslateExpression (scjExpr, transEnv)
           = TranslateExpression (e1, transEnv))
```

```
    ∨ (∃ e1: SCJExpression | scjExpr = unary e1
        • TranslateExpression (scjExpr, transEnv)
          = TranslateExpression (e1, transEnv))
```

```
ExtractParamComs: seq SCJExpression × TranslationEnv ⇸ seq Com
────────────────
∀ args: seq SCJExpression; transEnv: TranslationEnv
  • (∃ te: TypeElement; e: SCJExpression
       • head args = typeCast (te, e)
         ∧ ExtractParamComs (args, transEnv)
           = ExtractParamComs ((⟨e⟩ ⁀ tail args), transEnv))
    ∨ (∃ scje1, scje2: SCJExpression; c: Com
        • head args = binary (scje1, scje2)
          ∧ c
            = SimplifyCommandPair ((TranslateExpression (scje1,
                                                      transEnv)),
                                (TranslateExpression (scje2,
                                                      transEnv)))
          ∧ ¬ c = Skip
          ∧ ExtractParamComs (args, transEnv)
            = ⟨c⟩ ⁀ ExtractParamComs ((tail args), transEnv))
    ∨ (∃ te: TypeElement; scje: seq SCJExpression; c: Com
        • head args = newArray (te, scje)
          ∧ c = TranslateExpression ((head args), transEnv)
          ∧ ExtractParamComs (args, transEnv)
            = ⟨c⟩ ⁀ ExtractParamComs ((tail args), transEnv))
    ∨ (∃ name: Name; scje: seq SCJExpression; c: Com
        • head args = newClass (name, scje)
          ∧ c = TranslateExpression ((head args), transEnv)
          ∧ ExtractParamComs (args, transEnv)
            = ⟨c⟩ ⁀ ExtractParamComs ((tail args), transEnv))
    ∨ (∃ scje: SCJExpression; scjes: seq SCJExpression; name: Name; c: Com
        • ((head args = methodInvocation (scje, scjes)
           ∨ head args = memberSelect (scje, name))
          ∧ c = TranslateExpression ((head args), transEnv)
          ∧ (∃ mc: methodCall
               • (c = MethodCall mc
                  ∧ ExtractParamComs (args, transEnv)
                    = ⟨(GetMethodCallReturnDec (mc, c)).2⟩
                      ⁀ ExtractParamComs ((tail args), transEnv)))
          ∨ (∃ c1, c2: Com; mc: methodCall
               • c = Seq (c1, c2)
                 ∧ c2 = MethodCall mc
                 ∧ ExtractParamComs (args, transEnv)
                   = ⟨(GetMethodCallReturnDec (mc, c)).2⟩
                     ⁀ ExtractParamComs ((tail args), transEnv))))
```

```
ExtractParamExprs: seq SCJExpression × TranslationEnv ⇸ seq Expr
────────────────
∀ args: seq SCJExpression; transEnv: TranslationEnv
  • (∃ te: TypeElement; scje: SCJExpression
       • head args = typeCast (te, scje)
         ∧ ExtractParamExprs (args, transEnv)
           = ExtractParamExprs ((⟨scje⟩ ⁀ tail args), transEnv))
    ∨ (∃ scje1, scje2: SCJExpression; e: Expr
            • head args = binary (scje1, scje2)
             ∧ e = ExtractExpression (head args)
             ∧ ExtractParamExprs (args, transEnv)
                 = ⟨e⟩ ⁀ ExtractParamExprs ((tail args), transEnv))
    ∨ (∃ te: TypeElement; name: Name; scje: seq SCJExpression; c, c1,
        c2: Com; nI: newInstance; e: Expr
            • (head args = newArray (te, scje)
              ∨ head args = newClass (name, scje))
             ∧ c = TranslateExpression ((head args), transEnv)
             ∧ c = Seq (c1, c2)
             ∧ c2 = NewInstance nI
             ∧ ExtractParamExprs (args, transEnv)
                 = ⟨nI.le⟩ ⁀ ExtractParamExprs ((tail args), transEnv))
    ∨ (∃ scje: SCJExpression; scjes: seq SCJExpression; name: Name; c: Com;
        e: Expr
            • ((head args = methodInvocation (scje, scjes)
               ∨ head args = memberSelect (scje, name))
              ∧ c = TranslateExpression ((head args), transEnv)
              ∧ (∃ mc: methodCall
                    • (c = MethodCall mc
                      ∧ ExtractParamExprs (args, transEnv)
                          = ⟨(GetMethodCallReturnDec (mc, c)).1⟩
                            ⁀ ExtractParamExprs ((tail args), transEnv)))
              ∨ (∃ c1, c2: Com; mc: methodCall
                    • c = Seq (c1, c2)
                     ∧ c2 = MethodCall mc
                     ∧ ExtractParamExprs (args, transEnv)
                         = ⟨(GetMethodCallReturnDec (mc, c)).1⟩
                           ⁀ ExtractParamExprs ((tail args), transEnv))))
```

```
TranslateVariable: SCJVariable × SCJmSafeProgram × TranslationEnv ⇸ seq Com
─────────────
∀ scjVar: SCJVariable; program: SCJmSafeProgram; transEnv: TranslationEnv
   • ∃ dec: Dec; scjExpr: SCJExpression; com: Com
        • dec.var.name = scjVar.name
         ∧ dec.var.varType.type = scjVar.type.name
         ∧ scjExpr = scjVar.init
         ∧ dec.var ∈ transEnv.variables
         ∧ (scjExpr = null
           ∧ TranslateVariable (scjVar, program, transEnv) = ⟨Decl dec⟩
           ∨ (∃ type: TypeElement; e1: SCJExpression; var: SCJVariable
                 | scjExpr = typeCast (type, e1)
                  ∧ var.mods = scjVar.mods
                  ∧ var.type = scjVar.type
                  ∧ var.name = scjVar.name
                  ∧ var.init = e1
                • TranslateVariable (scjVar, program, transEnv)
                   = TranslateVariable (var, program, transEnv))
           ∨ ((∃ name: Name; scjArgs: seq SCJExpression; args: seq Expr;
                nI: newInstance
                  | scjExpr = newClass (name, scjArgs)
                   ∧ nI.le = ID (var dec.var)
                   ∧ nI.mrc = Current
                   ∧ nI.type = dec.var.varType
                   ∧ nI.args = ExtractParamExprs (scjArgs, transEnv)
```

```
            • dec.var.varType.isReference = True
              ∧ dec.var.varType.type = name
              ∧ com
                  = MergeSideEffectsParamsCom ((NewInstance nI),
                                (ExtractParamComs (scjArgs,
                                                transEnv)),
                                Skip))
        ∨ (∃ type: TypeElement; scjInits: seq SCJExpression;
            nI: newInstance
              | scjExpr = newArray (type, scjInits)
                ∧ nI.le = ID (var dec.var)
                ∧ nI.mrc = Current
                ∧ nI.type = dec.var.varType
                ∧ nI.args = ExtractParamExprs (scjInits, transEnv)
              • dec.var.varType.isReference = True
                ∧ dec.var.varType.isArray = True
                ∧ com
                    = MergeSideEffectsParamsCom ((NewInstance nI),
                                  (ExtractParamComs (scjInits,
                                                  transEnv)),
                                  Skip))
        ∨ (∃ lhs: SCJExpression; args: seq SCJExpression;
            lhsCom: Com
              | scjExpr = methodInvocation (lhs, args)
                ∧ lhsCom = TranslateExpression (lhs, transEnv)
              • (∃ c1, c2: Com | lhsCom = Seq (c1, c2)
                  • ((∃ le1: LExpr; n1: Name; seqExpr: seq Expr;
                      rexpr: Expr; sideEffect: Com;
                      mc: methodCall
                        | mc.le = le1
                          ∧ mc.name = n1
                          ∧ mc.args = seqExpr
                          ∧ c2 = MethodCall mc
                          ∧ (rexpr, sideEffect)
                              = GetMethodCallReturnDec (mc,
                                              lhsCom)
                        • com
                            = Seq (sideEffect,
                                  (Asgn ((ID (var dec.var)),
                                        rexpr))))
                    ∨ (∃ le1: LExpr; n1: Name;
                        seqExpr: seq Expr; rexpr: Expr;
                        sideEffect: Com; mc: methodCall
                          | mc.le = le1
                            ∧ mc.name = n1
                            ∧ mc.args = seqExpr
                            ∧ lhsCom = MethodCall mc
                            ∧ (rexpr, sideEffect)
                                = GetMethodCallReturnDec (mc,
                                                lhsCom)
                          • com
                              = Seq (sideEffect,
                                    (Asgn ((ID (var dec.var)),
                                          rexpr)))))))
        ∨ (∃ expr, index: SCJExpression
              | scjExpr = arrayAccess (expr, index)
```

```
• ((∃ e1: SCJExpression; args: seq SCJExpression; c1,
   c2, sideEffect: Com; rexpr: Expr
     | expr = methodInvocation (e1, args)
      ∧ c1 = TranslateExpression (expr, transEnv)
      ∧ c2
          = TranslateExpression (index, transEnv)
   • (∃ le1: LExpr; n1: Name; args2: seq Expr;
       c3: Com; mc: methodCall
         | mc.le = le1
          ∧ mc.name = n1
          ∧ mc.args = args2
          ∧ c1 = MethodCall mc
          ∧ (rexpr, c3)
              = GetMethodCallReturnDec (mc,
                                        c1)
          ∧ sideEffect
              = SimplifyCommandPair (c3, c2)
       • com
           = SimplifyCommandPair (sideEffect,
                         (Asgn ((ID (var dec.var)),
                                rexpr))))
      ∨ (∃ seq1, seq2: Com; c3: Com; le1: LExpr;
          n1: Name; args2: seq Expr;
          mc: methodCall
           | c1 = Seq (seq1, seq2)
            ∧ mc.le = le1
            ∧ mc.name = n1
            ∧ mc.args = args2
            ∧ seq2 = MethodCall mc
            ∧ (rexpr, c3)
                = GetMethodCallReturnDec (mc,
                                          c1)
            ∧ sideEffect
                = SimplifyCommandPair (c3, c2)
         • com
             = SimplifyCommandPair (sideEffect,
                           (Asgn ((ID (var dec.var)),
                                  rexpr)))))
  ∨ (∃ c1, c2, sideEffect: Com; rexpr: Expr
       | rexpr = ExtractExpression scjExpr
        ∧ c1
            = TranslateExpression (expr,
                           transEnv)
        ∧ c2
            = TranslateExpression (index,
                           transEnv)
     • sideEffect = SimplifyCommandPair (c1, c2)
        ∧ com
            = SimplifyCommandPair (sideEffect,
                          (Asgn ((ID (var dec.var)),
                                 rexpr))))))
∨ (∃ sideEffect: Com; rhs: Expr
     | sideEffect
        = TranslateExpression (scjExpr, transEnv)
      ∧ rhs = ExtractExpression scjExpr
  • (sideEffect = Skip
                    ∧ com = Asgn ((ID (var dec.var)), rhs)
                    ∨ (∃ le1: LExpr; n1: Name; seqExpr: seq Expr; mc1,
                       mc2: methodCall
                        | mc1.le = le1
                         ∧ mc1.name = n1
                         ∧ mc2.args = seqExpr
                         ∧ sideEffect = MethodCall mc1
                         ∧ mc2.le = le1
                         ∧ mc2.name = n1
                         ∧ mc2.args = seqExpr ⁀ ⟨ID (var dec.var)⟩
                        • com = MethodCall mc2)
                    ∨ com
                       = Seq (sideEffect,
                            (Asgn ((ID (var dec.var)), rhs)))))
                ∧ (static ∈ scjVar.mods.flags
                  ∧ dec ∈ program.static
                  ∧ com ∈ program.sInit
                  ∧ TranslateVariable (scjVar, program, transEnv)
                     = ⟨Skip⟩
                  ∨ static ∉ scjVar.mods.flags
                   ∧ TranslateVariable (scjVar, program, transEnv)
                      = ⟨Decl dec, com⟩)))
```

```
TranslateCommand: SCJCommand × SCJmSafeProgram × TranslationEnv ⇸ Com
────────────────────
∀ scjCom: SCJCommand; program: SCJmSafeProgram; transEnv: TranslationEnv
  | (scjCom, program, transEnv) ∈ dom TranslateCommand
   ∧ scjCom ∈ WellTypedComs
 • (∃ e1, e2: SCJExpression | scjCom = assert (e1, e2)
     • TranslateCommand (scjCom, program, transEnv) = Skip)
   ∨ (∃ bool: Boolean; c1: SCJCommand | scjCom = block (bool, c1)
       • TranslateCommand (scjCom, program, transEnv)
          = Scope (TranslateCommand (c1, program, transEnv)))
   ∨ (∃ name: Name | scjCom = break name
       • TranslateCommand (scjCom, program, transEnv) = Skip)
   ∨ (∃ name: Name | scjCom = continue name
       • TranslateCommand (scjCom, program, transEnv) = Skip)
   ∨ (∃ e1: SCJExpression; c1: SCJCommand | scjCom = doWhile (e1, c1)
       • TranslateCommand (scjCom, program, transEnv)
          = SimplifyCommandPair ((DoWhile ((TranslateCommand (c1,
                                                   program,
                                                   transEnv)),
                                       (ExtractExpression e1))),
                                  (TranslateExpression (e1, transEnv))))
   ∨ scjCom = empty ∧ TranslateCommand (scjCom, program, transEnv) = Skip
   ∨ (∃ mods: SCJModifier; type: TypeElement; name: Name; c1,
       c2: SCJCommand; e1: SCJExpression
       | scjCom = eFor (mods, type, name, c1, e1, c2)
       • TranslateCommand (scjCom, program, transEnv)
          = SimplifyCommandPair ((TranslateExpression (e1, transEnv)),
                                 (For ((TranslateCommand (c1, program,
                                                  transEnv)),
                                    (ExtractExpression e1), Skip,
                                    (TranslateCommand (c2, program,
                                                  transEnv))))))
```

```
∨ (∃ e1: SCJExpression | scjCom = expression e1
    • TranslateCommand (scjCom, program, transEnv)
      = TranslateExpression (e1, transEnv))
∨ (∃ c1, c2, c3: SCJCommand; e1: SCJExpression
    | scjCom = for (c1, e1, c2, c3)
    • TranslateCommand (scjCom, program, transEnv)
      = SimplifyCommandPair ((TranslateExpression (e1, transEnv)),
                  (For ((TranslateCommand (c1, program,
                                      transEnv)),
                     (ExtractExpression e1),
                     (TranslateCommand (c2, program,
                                      transEnv)),
                     (TranslateCommand (c3, program,
                                      transEnv))))))
∨ (∃ e1: SCJExpression; c1, c2: SCJCommand | scjCom = if (e1, c1, c2)
    • TranslateCommand (scjCom, program, transEnv)
      = SimplifyCommandPair ((TranslateExpression (e1, transEnv)),
                  (If ((ExtractExpression e1),
                     (TranslateCommand (c1, program,
                                      transEnv)),
                     (TranslateCommand (c2, program,
                                      transEnv))))))
∨ (∃ name: Name; c1: SCJCommand | scjCom = labeled (name, c1)
    • TranslateCommand (scjCom, program, transEnv)
      = TranslateCommand (c1, program, transEnv))
∨ (∃ e1: SCJExpression; lexpr: Expr; v: Variable
    | scjCom = return e1
      ∧ v.name = Result
      ∧ v.varType.resultVar = True
    • (e1 ≠ null
      ∧ ((∃ te: TypeElement; args: seq SCJExpression;
          nI: newInstance
            | e1 = newArray (te, args)
              ∧ v.varType.type = te.name
              ∧ nI.mrc = Current
              ∧ nI.le = ID (var v)
              ∧ nI.args = ExtractParamExprs (args, transEnv)
              ∧ nI.type = v.varType
            • TranslateCommand (scjCom, program, transEnv)
              = MergeSideEffectsParamsCom ((NewInstance nI),
                          (ExtractParamComs (args,
                                      transEnv)),
                          Skip))
        ∨ (∃ args: seq SCJExpression; nI: newInstance; name: Name
            | e1 = newClass (name, args)
              ∧ nI.type.type = name
              ∧ nI.type.isReference = True
              ∧ nI.le = ID (var v)
              ∧ nI.mrc = Current
              ∧ nI.args = ExtractParamExprs (args, transEnv)
              ∧ nI.type = v.varType
            • TranslateCommand (scjCom, program, transEnv)
              = MergeSideEffectsParamsCom ((NewInstance nI),
                          (ExtractParamComs (args,
                                      transEnv)),
                          Skip))
              ∨ (∃ le: SCJExpression; args: seq SCJExpression; c: Com
                  | e1 = methodInvocation (le, args)
                    ∧ c = TranslateExpression (e1, transEnv)
                  • ((∃ mc: methodCall; rexpr: Expr
                        | c = MethodCall mc
                          ∧ rexpr
                             = (GetMethodCallReturnDec (mc, c)).1
                        • TranslateCommand (scjCom, program,
                                   transEnv)
                           = Seq ((GetMethodCallReturnDec (mc,
                                              c)).2,
                               (Asgn ((ID (var v)), rexpr))))
                     ∨ (∃ c1, c2: Com; mc: methodCall; rexpr: Expr
                        | c = Seq (c1, c2)
                          ∧ c2 = MethodCall mc
                          ∧ rexpr
                             = (GetMethodCallReturnDec (mc, c)).1
                        • TranslateCommand (scjCom, program,
                                   transEnv)
                           = Seq ((GetMethodCallReturnDec (mc,
                                              c)).2,
                               (Asgn ((ID (var v)),
                                   rexpr)))))))
           ∨ e1 = null
             ∧ TranslateCommand (scjCom, program, transEnv) = Skip))
   ∨ (∃ a: SCJModifier; type: TypeElement; name: Name; c1: SCJCommand;
      c2: seq SCJCommand; e1: SCJExpression
      | scjCom = switch (a, type, name, c1, e1, c2)
      • (let translatedSeq ==
                    { i: 1 .. # c2
                      • i
                        ↦ TranslateCommand ((c2 i), program,
                                    transEnv) }
          • TranslateCommand (scjCom, program, transEnv)
             = SimplifyCommandPair ((TranslateExpression (e1,
                                          transEnv)),
                            (Switch ((ExtractExpression e1),
                                  translatedSeq)))))
   ∨ (∃ e1: SCJExpression; bool: Boolean; c1: SCJCommand
      | scjCom = synchronized1 (e1, bool, c1)
      • TranslateCommand (scjCom, program, transEnv)
         = TranslateCommand (c1, program, transEnv))
   ∨ (∃ e1: SCJExpression | scjCom = throw e1
      • TranslateCommand (scjCom, program, transEnv)
         = TranslateExpression (e1, transEnv))
   ∨ (∃ c1, c2: SCJCommand; eseq: seq SCJExpression; comseq: seq SCJCommand
      | scjCom = try (c1, eseq, comseq, c2)
      • (let translatedExprs ==
               { i: 1 .. # eseq • i ↦ ExtractExpression (eseq i) };
           translatedComs ==
                 { i: 1 .. # comseq
                   • i
                     ↦ TranslateCommand ((comseq i), program,
                                   transEnv) }
          • TranslateCommand (scjCom, program, transEnv)
             = Try ((TranslateCommand (c1, program, transEnv)),
                  translatedExprs, translatedComs,
                  (TranslateCommand (c2, program, transEnv)))))
    ∨ (∃ scjVar: SCJVariable | scjCom = variable scjVar
        • TranslateCommand (scjCom, program, transEnv)
          = CreateSingleCommand (TranslateVariable (scjVar, program,
                                      transEnv)))
    ∨ (∃ e1: SCJExpression; c1: SCJCommand | scjCom = while (e1, c1)
        • TranslateCommand (scjCom, program, transEnv)
          = Seq ((TranslateExpression (e1, transEnv)),
              (While ((ExtractExpression e1),
                  (TranslateCommand (c1, program, transEnv))))))
```

```
TranslateComponentsFieldsDecs: seq SCJClassComponent × SCJmSafeProgram ×
                 TranslationEnv
                 ⇸ seq Dec
───────────────
∀ components: seq SCJClassComponent; program: SCJmSafeProgram;
  transEnv: TranslationEnv
  • components = ⟨⟩
    ∧ TranslateComponentsFieldsDecs (components, program, transEnv) = ⟨⟩
    ∨ (∃ dec: Dec; h: SCJVariable; t: seq SCJClassComponent; seqCom: seq Com
        | components = ⟨ClassField h⟩ ⁀ t
          ∧ seqCom = TranslateVariable (h, program, transEnv)
          ∧ head seqCom = Decl dec
        • TranslateComponentsFieldsDecs (components, program, transEnv)
          = ⟨dec⟩
            ⁀ TranslateComponentsFieldsDecs (t, program, transEnv))
    ∨ (∃ h: SCJMethod; t: seq SCJClassComponent
        | components = ⟨ClassMethod h⟩ ⁀ t
        • TranslateComponentsFieldsDecs (components, program, transEnv)
          = ⟨⟩)
```

```
TranslateComponentsFieldsInits: seq SCJClassComponent × SCJmSafeProgram ×
                 TranslationEnv
                 ⇸ seq Com
───────────────
∀ components: seq SCJClassComponent; program: SCJmSafeProgram;
  transEnv: TranslationEnv
  • components = ⟨⟩
    ∧ TranslateComponentsFieldsInits (components, program, transEnv) = ⟨⟩
    ∨ (∃ h: SCJVariable; t: seq SCJClassComponent; seqCom: seq Com
        | components = ⟨ClassField h⟩ ⁀ t
          ∧ seqCom = TranslateVariable (h, program, transEnv)
        • TranslateComponentsFieldsInits (components, program, transEnv)
          = tail seqCom
            ⁀ TranslateComponentsFieldsInits (t, program, transEnv))
    ∨ (∃ h: SCJMethod; t: seq SCJClassComponent
        | components = ⟨ClassMethod h⟩ ⁀ t
        • TranslateComponentsFieldsInits (components, program, transEnv)
          = ⟨⟩)
```

```
TranslateParams: seq SCJVariable ⇸ seq Variable
───────────────
```

$$
\begin{array}{l}
\forall\ scjParams: \mathrm{seq}\ SCJVariable \\
\quad \bullet\ \exists\ params: \mathrm{seq}\ Variable \\
\qquad \bullet\ scjParams = \langle\rangle \land TranslateParams\ scjParams = \langle\rangle \\
\qquad \lor (\exists\ h: SCJVariable; t: \mathrm{seq}\ SCJVariable; v: Variable \\
\qquad\quad \mid\ scjParams = \langle h \rangle \frown t \\
\qquad\qquad \land v.name = h.name \\
\qquad\qquad \land v.varType.type = h.type.name \\
\qquad\quad \bullet\ TranslateParams\ scjParams = \langle v \rangle \frown TranslateParams\ t)
\end{array}
$$

$$
\begin{array}{l}
MethodResultParam: SCJMethod \rightarrowtail\!\!\!\!\!\!+\ \mathrm{seq}\ Variable \\
\hline
\forall\ scjMethod: SCJMethod \\
\quad \bullet\ \exists\ var: Variable \\
\qquad \bullet\ scjMethod.returnType.name \neq void \\
\qquad \land scjMethod.body \neq \langle\rangle \\
\qquad \land var.name = Result \\
\qquad \land var.varType.type = scjMethod.returnType.name \\
\qquad \land MethodResultParam\ scjMethod = \langle var \rangle \\
\qquad \lor (scjMethod.returnType.name = void \lor scjMethod.body = \langle\rangle) \\
\qquad\quad \land MethodResultParam\ scjMethod = \langle\rangle
\end{array}
$$

$$
\begin{array}{l}
TranslateCommandSeq: \mathrm{seq}\ SCJCommand \times SCJmSafeProgram \times TranslationEnv \\
\qquad\qquad \rightarrowtail\!\!\!\!\!\!+\ \mathrm{seq}\ Com \\
\hline
\forall\ seqCom: \mathrm{seq}\ SCJCommand; program: SCJmSafeProgram; transEnv: TranslationEnv \\
\quad \bullet\ TranslateCommandSeq\ (seqCom, program, transEnv) \\
\qquad = \langle TranslateCommand\ ((head\ seqCom), program, transEnv) \rangle \\
\qquad\ \frown TranslateCommandSeq\ ((tail\ seqCom), program, transEnv)
\end{array}
$$

$$
\begin{array}{l}
TranslateMethod: SCJMethod \times Name \times SCJmSafeProgram \times TranslationEnv \rightarrowtail\!\!\!\!\!\!+\ Method \\
\hline
\forall\ scjMethod: SCJMethod; name: Name; program: SCJmSafeProgram; \\
\ \ transEnv: TranslationEnv \\
\quad \bullet\ \exists\ method: Method \\
\qquad \mid\ scjMethod.name = method.name \\
\qquad \land method.class = name \\
\qquad \land method.params \\
\qquad\quad = TranslateParams\ scjMethod.params \\
\qquad\quad\ \frown MethodResultParam\ scjMethod \\
\qquad \land method.returnType = scjMethod.returnType.name \\
\qquad \land method.body \\
\qquad\quad = CreateSingleCommand\ (TranslateCommandSeq\ (scjMethod.body, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad program, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad transEnv)) \\
\qquad \bullet\ TranslateMethod\ (scjMethod, name, program, transEnv) = method
\end{array}
$$

$$
\begin{array}{l}
TranslateConstr: SCJMethod \times Name \times SCJmSafeProgram \times TranslationEnv \rightarrowtail\!\!\!\!\!\!+\ Method \\
\hline
\forall\ scjMethod: SCJMethod; name: Name; program: SCJmSafeProgram; \\
\ \ transEnv: TranslationEnv \\
\quad \bullet\ \exists\ method: Method \\
\qquad \mid\ scjMethod.name = method.name
\end{array}
$$

```
          $\land$ method.class = name
          $\land$ method.params = TranslateParams scjMethod.params
          $\land$ method.body
              = CreateSingleCommand (TranslateCommandSeq (scjMethod.body,
                                                          program,
                                                          transEnv))
      $\bullet$ TranslateConstr (scjMethod, name, program, transEnv) = method
```

```
TranslateClass: SCJClass $\times$ SCJmSafeProgram $\times$ TranslationEnv $\pfun$ Class
────────────────────────────
$\forall$ scjClass: SCJClass; program: SCJmSafeProgram; transEnv: TranslationEnv
  $\bullet$ $\exists$ class: Class
      $\bullet$ class.fields
          = TranslateComponentsFieldsDecs (scjClass.members, program,
                                           transEnv)
        $\land$ class.init
          = TranslateComponentsFieldsInits (scjClass.members, program,
                                            transEnv)
        $\land$ class.constrs
          = $\bigcup$ {  classComponent: ran scjClass.members
                $\bullet$ {  method: SCJMethod
                     | classComponent = ClassMethod method
                       $\land$ method.name = class.name
                     $\bullet$ (TranslateConstr (method, scjClass.name,
                                          program, transEnv)) } }
        $\land$ class.methods
          = $\bigcup$ {  classComponent: ran scjClass.members
                $\bullet$ {  method: SCJMethod
                     | classComponent = ClassMethod method
                       $\land$ method.name $\neq$ class.name
                     $\bullet$ (TranslateMethod (method, scjClass.name,
                                          program, transEnv)) } }
        $\land$ class.name = scjClass.name
        $\land$ TranslateClass (scjClass, program, transEnv) = class
```

```
TranslateHandler: SCJClass $\times$ SCJmSafeProgram $\times$ TranslationEnv $\pfun$ Handler
────────────────────────────
$\forall$ scjClass: SCJClass; program: SCJmSafeProgram; transEnv: TranslationEnv
  $\bullet$ $\exists$ handler: Handler
      $\bullet$ handler.fields
          = TranslateComponentsFieldsDecs (scjClass.members, program,
                                           transEnv)
        $\land$ handler.init
          = TranslateComponentsFieldsInits (scjClass.members, program,
                                            transEnv)
        $\land$ handler.constrs
          = $\bigcup$ {  classComponent: ran scjClass.members
                $\bullet$ {  method: SCJMethod
                     | classComponent = ClassMethod method
                       $\land$ method.name = handler.name
                     $\bullet$ (TranslateConstr (method, scjClass.name,
                                          program, transEnv)) } }
        $\land$ handler.methods
          = $\bigcup$ {  classComponent: ran scjClass.members
```

```
            • {  $method: SCJMethod$
                 | $classComponent = ClassMethod\ method$
                  $\wedge\ method.name \neq handler.name$
                  $\wedge\ method.name \neq handleEvent$
                 • $(TranslateMethod\ (method, scjClass.name,$
                              $program, transEnv))$ } }
   $\wedge\ (\exists\ classComponent$: ran $scjClass.members; method: SCJMethod$
       | $classComponent = ClassMethod\ method$
        $\wedge\ method.name = handleEvent$
      • $handler.hAe$
         $= CreateSingleCommand\ (TranslateCommandSeq\ (method.body,$
                                    $program,$
                                    $transEnv)))$
   $\wedge\ handler.name = scjClass.name$
   $\wedge\ TranslateHandler\ (scjClass, program, transEnv) = handler$
```

```
$Handlers: SCJProgram \rightarrowtail \mathbb{P}\ Name$
---------------------------------------
$\forall\ program: SCJProgram$
  • $Handlers\ program$
     = {  $c: program.classes$
          | $c.extends = PeriodicEventHandler$
           $\vee\ c.extends = AperiodicEventHandler$ •  $c.name$ }
```

```
$AnalyseHandlers: SCJProgram \times SCJClass \rightarrowtail \mathbb{P}\ Name$
---------------------------------------
$\forall\ program: SCJProgram; class: SCJClass$
  • $AnalyseHandlers\ (program, class)$
     $= \bigcup$ {  $m$: ran $class.members; method: SCJMethod$
          | $m = ClassMethod\ method \wedge method.name = initialize$
          • {  $c$: ran $method.body; e: SCJExpression; n: Name;$
               $args$: seq $SCJExpression$
               | $c = expression\ e$
                $\wedge\ e = newClass\ (n, args)$
                $\wedge\ n \in Handlers\ program$ •  $n$ } }
```

```
$TranslateMission: SCJClass \times SCJProgram \times SCJmSafeProgram \times TranslationEnv$
          $\rightarrowtail Mission$
---------------------------------------
$\forall\ scjClass: SCJClass; scjProg: SCJProgram; program: SCJmSafeProgram;$
 $transEnv: TranslationEnv$
  • $\exists\ mission: Mission$
     • **let** $missionMethods$ == {$initialize, cleanUp$}
       • $mission.fields$
          $= TranslateComponentsFieldsDecs\ (scjClass.members, program,$
                              $transEnv)$
        $\wedge\ mission.init$
          $= TranslateComponentsFieldsInits\ (scjClass.members,$
                              $program, transEnv)$
        $\wedge\ mission.constrs$
          $= \bigcup$ {  $classComponent$: ran $scjClass.members$
               • {  $method: SCJMethod$
                    | $classComponent = ClassMethod\ method$
                    ∧ method.name = mission.name
                  • (TranslateConstr (method, scjClass.name,
                                      program,
                                      transEnv)) } }
        ∧ mission.handlers = AnalyseHandlers (scjProg, scjClass)
        ∧ mission.methods
            = ⋃ {  classComponent: ran scjClass.members
                  • {  method: SCJMethod
                        | classComponent = ClassMethod method
                          ∧ method.name ≠ mission.name
                          ∧ method.name ∉ missionMethods
                        • (TranslateMethod (method, scjClass.name,
                                            program,
                                            transEnv)) } }
        ∧ (∃ classComponent: ran scjClass.members; method: SCJMethod
            | classComponent = ClassMethod method
              ∧ method.name = initialize
            • mission.initialize
                = CreateSingleCommand (TranslateCommandSeq (method.body,
                                                            program,
                                                            transEnv)))
        ∧ (∃ classComponent: ran scjClass.members; method: SCJMethod
            | classComponent = ClassMethod method
              ∧ method.name = cleanUp
            • mission.cleanUp
                = CreateSingleCommand (TranslateCommandSeq (method.body,
                                                            program,
                                                            transEnv)))
        ∧ mission.name = scjClass.name
        ∧ TranslateMission (scjClass, scjProg, program, transEnv)
            = mission
```

```
| Missions: SCJProgram ⇸ ℙ Name
|------------------------------
| ∀ program: SCJProgram
|   • Missions program
|       = {  c: program.classes | c.extends = mission • c.name  }
```

```
| AnalyseMissions: SCJProgram × SCJClass ⇸ ℙ Name
|------------------------------
| ∀ program: SCJProgram; class: SCJClass
|   • AnalyseMissions (program, class)
|       = ⋃ {  m: ran class.members; method: SCJMethod
|             | m = ClassMethod method ∧ method.name = getNextMission
|             • {  c: ran method.body; e: SCJExpression; n: Name;
|                  args: seq SCJExpression
|                   | c = expression e
|                     ∧ e = newClass (n, args)
|                     ∧ n ∈ Missions program •  n  } }
```

```
| TranslateMissionSeq: SCJClass × SCJProgram × SCJmSafeProgram × TranslationEnv
|                    ⇸ MissionSeq
|------------------------------
```

```
∀ scjClass: SCJClass; scjProg: SCJProgram; program: SCJmSafeProgram;
 transEnv: TranslationEnv
  • ∃ missionSeq: MissionSeq
     • missionSeq.fields
        = TranslateComponentsFieldsDecs (scjClass.members, program,
                                         transEnv)
       ∧ missionSeq.init
          = TranslateComponentsFieldsInits (scjClass.members, program,
                                            transEnv)
       ∧ missionSeq.constrs
          = ∪ {  classComponent: ran scjClass.members
               • {  method: SCJMethod
                    | classComponent = ClassMethod method
                     ∧ method.name = missionSeq.name
                    • (TranslateConstr (method, scjClass.name,
                                        program, transEnv)) } }
       ∧ (∃ classComponent: ran scjClass.members; method: SCJMethod
            | classComponent = ClassMethod method
             ∧ method.name = getNextMission
            • missionSeq.getNextMission
               = CreateSingleCommand (TranslateCommandSeq (method.body,
                                                           program,
                                                           transEnv)))
       ∧ missionSeq.missions = AnalyseMissions (scjProg, scjClass)
       ∧ missionSeq.methods
          = ∪ {  classComponent: ran scjClass.members
               • {  method: SCJMethod
                    | classComponent = ClassMethod method
                     ∧ method.name ≠ missionSeq.name
                     ∧ method.name ≠ getNextMission
                    • (TranslateMethod (method, scjClass.name,
                                        program, transEnv)) } }
       ∧ missionSeq.name = scjClass.name
       ∧ TranslateMissionSeq (scjClass, scjProg, program, transEnv)
          = missionSeq
```

```
TranslateSafelet: SCJClass × SCJmSafeProgram × TranslationEnv ⇸ Safelet
─────────────────────────────
∀ scjClass: SCJClass; program: SCJmSafeProgram; transEnv: TranslationEnv
  • ∃ safelet: Safelet
     • let safeletMethods == {initializeApplication, getSequencer}
        • safelet.fields
           = TranslateComponentsFieldsDecs (scjClass.members, program,
                                            transEnv)
          ∧ safelet.init
             = TranslateComponentsFieldsInits (scjClass.members,
                                               program, transEnv)
          ∧ safelet.constrs
             = ∪ {  classComponent: ran scjClass.members
                  • {  method: SCJMethod
                       | classComponent = ClassMethod method
                        ∧ method.name = safelet.name
                       • (TranslateConstr (method, scjClass.name,
                                           program,
                                           transEnv)) } }
```

```
    ∧ safelet.methods
        = ⋃ {  classComponent: ran scjClass.members
               • {  method: SCJMethod
                    | classComponent = ClassMethod method
                      ∧ method.name ≠ safelet.name
                      ∧ method.name ∉ safeletMethods
                    • (TranslateMethod (method, scjClass.name,
                                        program,
                                        transEnv)) } }
    ∧ (∃ classComponent: ran scjClass.members; method: SCJMethod
        | classComponent = ClassMethod method
          ∧ method.name = initializeApplication
        • safelet.initializeApplication
            = CreateSingleCommand (TranslateCommandSeq (method.body,
                                                        program,
                                                        transEnv)))
    ∧ (∃ classComponent: ran scjClass.members; method: SCJMethod
        | classComponent = ClassMethod method
          ∧ method.name = getSequencer
        • safelet.getSequencer
            = CreateSingleCommand (TranslateCommandSeq (method.body,
                                                        program,
                                                        transEnv)))
    ∧ safelet.name = scjClass.name
    ∧ TranslateSafelet (scjClass, program, transEnv) = safelet
```

```
ExtractParamTypes: seq SCJVariable ⇸ seq Name
────────────────────
∀ vars: seq SCJVariable
  • ∃ var: SCJVariable | var = head vars
      • ExtractParamTypes vars
          = ⟨var.type.name⟩ ⁀ ExtractParamTypes (tail vars)
```

```
AnalyseMethodSig: SCJClass × SCJMethod ⇸ MethodSig
────────────────────
∀ class: SCJClass; method: SCJMethod
  • ∃ ms: MethodSig
      | ms.name = method.name
        ∧ ms.class = class.name
        ∧ ms.classExtends = class.extends
        ∧ ms.descendants = ∅
        ∧ ms.returnTypeName = method.returnType.name
        ∧ ms.paramTypes = ExtractParamTypes method.params
      • AnalyseMethodSig (class, method) = ms
```

```
AnalyseMethodSigsClass: SCJClass × seq SCJClassComponent ⇸ ℙ MethodSig
────────────────────
∀ class: SCJClass; components: seq SCJClassComponent
  • (∃ method: SCJMethod | head components = ClassMethod method
      • AnalyseMethodSigsClass (class, components)
          = {AnalyseMethodSig (class, method)}
            ∪ AnalyseMethodSigsClass (class, (tail components)))
    ∨ (∃ method: SCJMethod | head components ≠ ClassMethod method
        • AnalyseMethodSigsClass (class, components)
          = AnalyseMethodSigsClass (class, (tail components)))
```

```
CalculateDescendants: SCJProgram × ℙ MethodSig ⇸ ℙ MethodSig
──────────────────────────────
∀ program: SCJProgram; methods: ℙ MethodSig
  • CalculateDescendants (program, methods)
     = ⋃ {  ms: methods
           • {  c1: program.classes; ms2: MethodSig
               | ms.class = c1.name
                ∧ ms.name = ms2.name
                ∧ ms.class = ms2.class
                ∧ ms.classExtends = ms2.classExtends
                ∧ ms.returnType = ms2.returnType
                ∧ ms.returnTypeName = ms2.returnTypeName
                ∧ ms.paramTypes = ms2.paramTypes
                ∧ ms2.descendants
                   = {  c2: program.classes
                       | c1 ≠ c2 ∧ c2.extends = c1.name
                       • c2.name  } •  ms2  } }
```

```
AnalyseMethodSigs: SCJProgram ⇸ ℙ MethodSig
──────────────────────────────
∀ program: SCJProgram
  • AnalyseMethodSigs program
     = CalculateDescendants (program,
                     (⋃ {  c: program.classes
                          • (AnalyseMethodSigsClass (c,
                                          c.members))  }))
```

```
Extends: SCJClass × Name × SCJProgram ⇸ Boolean
──────────────────────────────
∀ class: SCJClass; name: Name; program: SCJProgram
  • class.extends ≠ Empty
    ∧ (class.extends = name ∧ Extends (class, name, program) = True
      ∨ (∃ c1: SCJClass | c1 ∈ program.classes ∧ c1.name = class.extends
         • Extends (class, name, program)
           = Extends (c1, name, program)))
    ∨ class.extends = Empty ∧ Extends (class, name, program) = False
```

```
Translate: SCJProgram ⇸ SCJmSafeProgram
──────────────────────────────
∀ program: dom Translate | program ∈ WellTypedProgs
  • ∃ scjmsafe: SCJmSafeProgram; transEnv: TranslationEnv
     | transEnv.methods = AnalyseMethodSigs program
     • (∃ scjSafelet: SCJClass
         | scjSafelet ∈ program.classes
          ∧ Extends (scjSafelet, safelet, program) = True
         • scjmsafe.safelet
           = TranslateSafelet (scjSafelet, scjmsafe, transEnv))
       ∧ (∃ scjMissionSeq: SCJClass
           | scjMissionSeq ∈ program.classes
          ∧ Extends (scjMissionSeq, missionSequencer, program)
              = True
        • scjmsafe.missionSeq
            = TranslateMissionSeq (scjMissionSeq, program,
                                   scjmsafe, transEnv))
  ∧ (∀ scjMission: SCJClass
        | scjMission ∈ program.classes
          ∧ Extends (scjMission, mission, program) = True
        • TranslateMission (scjMission, program, scjmsafe,
                            transEnv) ∈ scjmsafe.missions)
  ∧ (∀ scjHandler: SCJClass
        | scjHandler ∈ program.classes
          ∧ (Extends (scjHandler, PeriodicHandler, program) = True
            ∨ Extends (scjHandler, APeriodicHandler, program)
                = True)
        • TranslateHandler (scjHandler, scjmsafe, transEnv)
            ∈ scjmsafe.handlers)
  ∧ (∀ scjClass: SCJClass
        | scjClass ∈ program.classes
          ∧ abstract ∉ scjClass.modifiers.flags
          ∧ Extends (scjClass, safelet, program) = False
          ∧ Extends (scjClass, missionSequencer, program) = False
          ∧ Extends (scjClass, mission, program) = False
          ∧ Extends (scjClass, APeriodicHandler, program) = False
          ∧ Extends (scjClass, PeriodicHandler, program) = False
        • TranslateClass (scjClass, scjmsafe, transEnv)
            ∈ scjmsafe.classes)
  ∧ Translate program = scjmsafe
```

Appendix E

Checking technique in Z

$$ExprShareRelation == LExpr \leftrightarrow LExpr$$

$$ExprRefSet == LExpr \rightarrowtail \mathbb{P}\ RefCon$$

$$\begin{array}{l}
Env == \\
\quad \{\ env: ExprShareRelation \times ExprRefSet \\
\qquad \mid \forall\ rel, crel: ExprShareRelation; ref: ExprRefSet \\
\qquad\quad \mid (rel, ref) = env \land crel = rel^{*} \cup (rel^{*})^{\sim} \\
\qquad\quad \bullet\ \mathrm{dom}\ crel = \mathrm{dom}\ ref \\
\qquad\qquad \land (\forall\ e_1, e_2: LExpr \mid\ e_1 \mapsto e_2 \in crel \bullet\ ref\ e_1 = ref\ e_2)\ \}
\end{array}$$

$$MethodRefSet == LExpr \rightarrowtail \mathbb{P}\ MetaRefCon$$

$$\begin{array}{l}
MethodProperties == \\
\quad \{\ properties: ExprShareRelation \times MethodRefSet \\
\qquad \mid \forall\ rel, crel: ExprShareRelation; ref: MethodRefSet \\
\qquad\quad \mid (rel, ref) = properties \land crel = rel^{*} \cup (rel^{*})^{\sim} \\
\qquad\quad \bullet\ \mathrm{dom}\ crel = \mathrm{dom}\ ref \\
\qquad\qquad \land (\forall\ e_1, e_2: LExpr \mid\ e_1 \mapsto e_2 \in crel \bullet\ ref\ e_1 = ref\ e_2)\ \}
\end{array}$$

$$\begin{array}{|l}
PrefixOf: Expr \times Expr \rightarrowtail Boolean \\
\hline
\forall\ le_1, le_2: Expr \\
\quad \bullet\ le_1 = le_2 \land PrefixOf\,(le_1, le_2) = True \\
\qquad \lor (le_1 \neq le_2 \\
\qquad\quad \land ((le_1 = Null \lor le_1 = This \lor le_1 = Val \lor le_2 = Val) \\
\qquad\qquad \land PrefixOf\,(le_1, le_2) = False) \\
\qquad\quad \lor (\exists\ id: Identifier; fa: FieldAccess \mid\ le_1 = ID\ id \land le_2 = FA\ fa \\
\qquad\qquad\quad \bullet\ (\langle id \rangle\ \mathrm{prefix}\ fa \land PrefixOf\,(le_1, le_2) = True \\
\qquad\qquad\qquad \lor \neg\ \langle id \rangle\ \mathrm{prefix}\ fa \land PrefixOf\,(le_1, le_2) = False)) \\
\qquad\quad \lor (\exists\ fa_1, fa_2: FieldAccess \mid\ le_1 = FA\ fa_1 \land le_2 = FA\ fa_2 \\
\qquad\qquad\quad \bullet\ (fa_2\ \mathrm{prefix}\ fa_1 \land PrefixOf\,(le_1, le_2) = True \\
\qquad\qquad\qquad \lor \neg\ fa_2\ \mathrm{prefix}\ fa_1 \land PrefixOf\,(le_1, le_2) = False)))
\end{array}$$

$$\begin{array}{|l}
LengthOf: LExpr \rightarrowtail \mathbb{N} \\
\hline
\forall\ e: LExpr \\
\quad \bullet\ \exists\ fa: FieldAccess \\
\qquad \bullet\ e = FA\ fa \land LengthOf\ e = \#\ fa \lor e \neq FA\ fa \land LengthOf\ e = 1
\end{array}$$

$$\begin{array}{|l}
FieldOf: Expr \times Expr \rightarrowtail Boolean \\
\hline
\forall\ le_1, le_2: Expr \\
\quad \bullet\ le_1 = le_2 \land FieldOf\,(le_1, le_2) = False \\
\qquad \lor (le_1 \neq le_2 \\
\qquad\quad \land ((le_1 = Null \lor le_1 = This \lor le_1 = Val \lor le_2 = Val) \\
\qquad\qquad \land FieldOf\,(le_1, le_2) = False) \\
\qquad\quad \lor (\exists\ id: Identifier; fa: FieldAccess \mid\ le_1 = ID\ id \land le_2 = FA\ fa \\
\qquad\qquad\quad \bullet\ (\langle id \rangle\ \mathrm{prefix}\ fa \\
\qquad\qquad\qquad \land LengthOf\ le_2 = LengthOf\ le_1 + 1 \\
\qquad\qquad\qquad \land FieldOf\,(le_2, le_1) = True \\
\qquad\qquad\qquad \lor \neg\ \langle id \rangle\ \mathrm{prefix}\ fa \land FieldOf\,(le_2, le_1) = False)) \\
\qquad\quad \lor (\exists\ fa_1, fa_2: FieldAccess \mid\ le_1 = FA\ fa_1 \land le_2 = FA\ fa_2 \\
\qquad\qquad\quad \bullet\ (fa_1\ \mathrm{prefix}\ fa_2
\end{array}$$

$\quad\quad \land\ LengthOf\ le_2 = LengthOf\ le_1 + 1$
$\quad\quad \land\ FieldOf\ (le_2, le_1) = True$
$\quad\quad \lor\ \neg\ fa_2\ \text{prefix}\ fa_1 \land FieldOf\ (le_2, le_1) = False)))$

$ExprShareAdd: LExpr \times LExpr \times ExprShareRelation \rightarrowtail ExprShareRelation$

$\forall\ le_1, le_2: LExpr; rel: ExprShareRelation \mid\ PrefixOf\ (le_1, le_2) = False$
$\quad \bullet\ ExprShareAdd\ (le_1, le_2, rel) = (rel \cup \{(le_1 \mapsto le_2), (le_2 \mapsto le_1)\})^{*}$

$ExprShareAddSet: \mathbb{P}\ (LExpr \times LExpr) \times ExprShareRelation \rightarrowtail ExprShareRelation$

$\forall\ set: \mathbb{P}\ (LExpr \times LExpr); rel: ExprShareRelation$
$\quad \bullet\ ExprShareAddSet\ (set, rel)$
$\quad\quad = (rel$
$\quad\quad\quad \cup \bigcup \{\ le_1, le_2: LExpr$
$\quad\quad\quad\quad \mid\ (le_1, le_2) \in set \land PrefixOf\ (le_1, le_2) = False$
$\quad\quad\quad\quad \bullet\ \{(le_1 \mapsto le_2), (le_2 \mapsto le_1)\}\ \})^{*}$

$ExprShareAddEnv: LExpr \times LExpr \times Env \rightarrowtail Env$

$\forall\ le_1, le_2: LExpr; env: Env$
$\quad \bullet\ \exists\ rel: ExprShareRelation; ref: ExprRefSet \mid\ env = (rel, ref)$
$\quad\quad \bullet\ ExprShareAddEnv\ (le_1, le_2, env)$
$\quad\quad\quad = ExprShareAdd\ (le_1, le_2, rel) \mapsto ref$

$ExprShareAddSetEnv: \mathbb{P}\ (LExpr \times LExpr) \times Env \rightarrowtail Env$

$\forall\ set: \mathbb{P}\ (LExpr \times LExpr); env: Env$
$\quad \bullet\ \exists\ rel: ExprShareRelation; ref: ExprRefSet \mid\ env = (rel, ref)$
$\quad\quad \bullet\ ExprShareAddSetEnv\ (set, env)$
$\quad\quad\quad = (rel$
$\quad\quad\quad\quad \cup \bigcup \{\ le_1, le_2: LExpr$
$\quad\quad\quad\quad\quad \mid\ (le_1, le_2) \in set \land PrefixOf\ (le_1, le_2) = False$
$\quad\quad\quad\quad\quad \bullet\ \{(le_1 \mapsto le_2), (le_2 \mapsto le_1)\}\ \})^{*}$
$\quad\quad\quad \mapsto ref$

$ExprShareRemove: LExpr \times ExprShareRelation \rightarrowtail ExprShareRelation$

$\forall\ le: LExpr; rel: ExprShareRelation$
$\quad \bullet\ \textbf{let}\ toRemove\ == \{\ e: \text{dom}\ rel \mid\ e = le \lor PrefixOf\ (le, e) = True \bullet\ e\ \}$
$\quad\quad \bullet\ ExprShareRemove\ (le, rel) = toRemove \lhd rel \rhd toRemove$

$ExprShareRemoveSet: \mathbb{P}\ LExpr \times ExprShareRelation \rightarrowtail ExprShareRelation$

$\forall\ lexprs: \mathbb{P}\ LExpr; rel: ExprShareRelation$
$\quad \bullet\ \textbf{let}\ toRemove\ ==$
$\quad\quad\quad\quad \bigcup \{\ le: lexprs$
$\quad\quad\quad\quad\quad \bullet\ \{\ e: \text{dom}\ rel \mid\ e = le \lor PrefixOf\ (le, e) = True$
$\quad\quad\quad\quad\quad\quad \bullet\ e\ \}\ \}$
$\quad\quad \bullet\ ExprShareRemoveSet\ (lexprs, rel) = toRemove \lhd rel \rhd toRemove$

$$
\begin{array}{|l}
\mathit{ExprShareRemoveEnv}: \mathit{LExpr} \times \mathit{Env} \rightarrowtail \mathit{Env} \\
\hline
\forall\, \mathit{le}: \mathit{LExpr};\ \mathit{env}: \mathit{Env} \\
\quad \bullet\ \exists\, \mathit{rel}: \mathit{ExprShareRelation};\ \mathit{ref}: \mathit{ExprRefSet} \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet\ \mathit{ExprShareRemoveEnv}\ (\mathit{le}, \mathit{env}) = \mathit{ExprShareRemove}\ (\mathit{le}, \mathit{rel}) \mapsto \mathit{ref}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprRefAdd}: \mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon} \times \mathit{ExprRefSet} \rightarrowtail \mathit{ExprRefSet} \\
\hline
\forall\, \mathit{le}: \mathit{LExpr};\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon};\ \mathit{ref}: \mathit{ExprRefSet} \\
\quad \bullet\ \mathit{ExprRefAdd}\ (\mathit{le}, \mathit{rcs}, \mathit{ref}) = \mathit{ref} \oplus \{(\mathit{le} \mapsto \mathit{rcs})\}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprRefUpdate}: \mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon} \times \mathit{ExprRefSet} \rightarrowtail \mathit{ExprRefSet} \\
\hline
\forall\, \mathit{le}: \mathit{LExpr};\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon};\ \mathit{ref}: \mathit{ExprRefSet} \\
\quad \bullet\ \mathit{ExprRefUpdate}\ (\mathit{le}, \mathit{rcs}, \mathit{ref}) = \mathit{ref} \oplus \{(\mathit{le} \mapsto \mathit{ref}\ \mathit{le} \cup \mathit{rcs})\}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprRefUpdateSet}: \mathbb{P}\ (\mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon}) \times \mathit{ExprRefSet} \rightarrowtail \mathit{ExprRefSet} \\
\hline
\forall\, \mathit{set}: \mathbb{P}\ (\mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon});\ \mathit{ref}: \mathit{ExprRefSet} \\
\quad \bullet\ \mathit{ExprRefUpdateSet}\ (\mathit{set}, \mathit{ref}) \\
\qquad = \mathit{ref} \\
\qquad\quad \oplus \{\ \mathit{le}: \mathit{LExpr};\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon} \mid (\mathit{le}, \mathit{rcs}) \in \mathit{set} \\
\qquad\qquad \bullet\ (\mathit{le} \mapsto \mathit{ref}\ \mathit{le} \cup \mathit{rcs})\ \}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprRefAddEnv}: \mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon} \times \mathit{Env} \rightarrowtail \mathit{Env} \\
\hline
\forall\, \mathit{le}: \mathit{LExpr};\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon};\ \mathit{env}: \mathit{Env} \\
\quad \bullet\ \exists\, \mathit{rel}: \mathit{ExprShareRelation};\ \mathit{ref}: \mathit{ExprRefSet} \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet\ \mathit{ExprRefAddEnv}\ (\mathit{le}, \mathit{rcs}, \mathit{env}) = \mathit{rel} \mapsto \mathit{ExprRefAdd}\ (\mathit{le}, \mathit{rcs}, \mathit{ref})
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprRefUpdateEnv}: \mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon} \times \mathit{Env} \rightarrowtail \mathit{Env} \\
\hline
\forall\, \mathit{le}: \mathit{LExpr};\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon};\ \mathit{env}: \mathit{Env} \\
\quad \bullet\ \exists\, \mathit{rel}: \mathit{ExprShareRelation};\ \mathit{ref}: \mathit{ExprRefSet} \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet\ \mathit{ExprRefUpdateEnv}\ (\mathit{le}, \mathit{rcs}, \mathit{env}) \\
\qquad\quad = \mathit{rel} \mapsto \mathit{ExprRefUpdate}\ (\mathit{le}, \mathit{rcs}, \mathit{ref})
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprRefUpdateSetEnv}: \mathbb{P}\ (\mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon}) \times \mathit{Env} \rightarrowtail \mathit{Env} \\
\hline
\forall\, \mathit{set}: \mathbb{P}\ (\mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon});\ \mathit{env}: \mathit{Env} \\
\quad \bullet\ \exists\, \mathit{rel}: \mathit{ExprShareRelation};\ \mathit{ref}: \mathit{ExprRefSet} \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet\ \mathit{ExprRefUpdateSetEnv}\ (\mathit{set}, \mathit{env}) \\
\qquad\quad = \mathit{rel} \\
\qquad\quad\ \mapsto \mathit{ref} \\
\qquad\qquad \oplus \{\ \mathit{le}: \mathit{LExpr};\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon} \mid (\mathit{le}, \mathit{rcs}) \in \mathit{set} \\
\qquad\qquad\quad \bullet\ (\mathit{le} \mapsto \mathit{ref}\ \mathit{le} \cup \mathit{rcs})\ \}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{ExprUpdateSetEnv}: \mathbb{P}\ (\mathit{LExpr} \times \mathit{LExpr}) \times \mathbb{P}\ (\mathit{LExpr} \times \mathbb{P}\ \mathit{RefCon}) \times \mathit{Env} \rightarrowtail \mathit{Env} \\
\hline
\end{array}
$$

$\forall\ shares: \mathbb{P}\ (LExpr \times LExpr);\ refSets: \mathbb{P}\ (LExpr \times \mathbb{P}\ RefCon);\ env: Env$
$\quad \bullet\ \exists\ rel: ExprShareRelation;\ ref: ExprRefSet \mid env = (rel, ref)$
$\quad\quad \bullet\ ExprUpdateSetEnv\ (shares, refSets, env)$
$\quad\quad\quad = (rel$
$\quad\quad\quad\quad \cup \bigcup \{\ le_1, le_2: LExpr \mid (le_1, le_2) \in shares$
$\quad\quad\quad\quad\quad \bullet\ \{(le_1 \mapsto le_2), (le_2 \mapsto le_1)\}\ \})^*$
$\quad\quad\quad \mapsto ref$
$\quad\quad\quad\quad \oplus \{\ le: LExpr;\ rcs: \mathbb{P}\ RefCon \mid (le, rcs) \in refSets$
$\quad\quad\quad\quad\quad \bullet\ (le \mapsto ref\ le \cup rcs)\ \}$

$ExprRefRemove: LExpr \times ExprRefSet \rightarrow\!\!\!\!\!+\ ExprRefSet$

$\forall\ le: LExpr;\ ref: ExprRefSet$
$\quad \bullet\ \mathbf{let}\ toRemove == \{\ e: \mathrm{dom}\ ref \mid e = le \lor PrefixOf\ (le, e) = True \bullet e\ \}$
$\quad\quad \bullet\ ExprRefRemove\ (le, ref) = toRemove \lhd ref$

$ExprRefRemoveSet: \mathbb{P}\ LExpr \times ExprRefSet \rightarrow\!\!\!\!\!+\ ExprRefSet$

$\forall\ lexprs: \mathbb{P}\ LExpr;\ ref: ExprRefSet$
$\quad \bullet\ \mathbf{let}\ toRemove ==$
$\quad\quad\quad\quad \bigcup \{\ le: lexprs$
$\quad\quad\quad\quad\quad \bullet\ \{\ e: \mathrm{dom}\ ref \mid e = le \lor PrefixOf\ (le, e) = True$
$\quad\quad\quad\quad\quad\quad \bullet\ e\ \}\ \}$
$\quad\quad \bullet\ ExprRefRemoveSet\ (lexprs, ref) = toRemove \lhd ref$

$ExprRefRemoveEnv: LExpr \times Env \rightarrow\!\!\!\!\!+\ Env$

$\forall\ le: LExpr;\ env: Env$
$\quad \bullet\ \exists\ rel: ExprShareRelation;\ ref: ExprRefSet \mid env = (rel, ref)$
$\quad\quad \bullet\ ExprRefRemoveEnv\ (le, env) = rel \mapsto ExprRefRemove\ (le, ref)$

$RemoveExprEnv: LExpr \times Env \rightarrow\!\!\!\!\!+\ Env$

$\forall\ le: LExpr;\ env: Env$
$\quad \bullet\ \exists\ rel: ExprShareRelation;\ ref: ExprRefSet \mid env = (rel, ref)$
$\quad\quad \bullet\ RemoveExprEnv\ (le, env)$
$\quad\quad\quad = ExprShareRemove\ (le, rel) \mapsto ExprRefRemove\ (le, ref)$

$RemoveExprSetEnv: \mathbb{P}\ LExpr \times Env \rightarrow\!\!\!\!\!+\ Env$

$\forall\ lexprs: \mathbb{P}\ LExpr;\ env: Env$
$\quad \bullet\ \exists\ rel: ExprShareRelation;\ ref: ExprRefSet \mid env = (rel, ref)$
$\quad\quad \bullet\ RemoveExprSetEnv\ (lexprs, env)$
$\quad\quad\quad = ExprShareRemoveSet\ (lexprs, rel)$
$\quad\quad\quad \mapsto ExprRefRemoveSet\ (lexprs, ref)$

$ExprRefJoin: ExprRefSet \times ExprRefSet \rightarrow\!\!\!\!\!+\ ExprRefSet$

$\forall\ ref_1, ref_2: ExprRefSet$
$\quad \bullet\ ExprRefJoin\ (ref_1, ref_2)$

$$\begin{array}{l}
\quad = (\mathrm{dom}\ ref_1 \setminus \mathrm{dom}\ ref_2) \lhd ref_1 \cup (\mathrm{dom}\ ref_2 \setminus \mathrm{dom}\ ref_1) \lhd ref_2 \\
\quad\ \cup \{\ le: \mathrm{dom}\ ref_1 \cap \mathrm{dom}\ ref_2 \bullet\ (le \mapsto ref_1\ le \cup ref_2\ le)\ \}
\end{array}$$

$$\begin{array}{|l}
DistExprRefJoin: \mathbb{F}\ ExprRefSet \pfun ExprRefSet \\
\hline
\forall erefsets: \mathbb{F}\ ExprRefSet \\
\quad \bullet\ \exists ref: erefsets \\
\qquad \bullet\ erefsets \neq \varnothing \\
\qquad\ \ \wedge DistExprRefJoin\ erefsets \\
\qquad\qquad = ExprRefJoin\ (ref, (DistExprRefJoin\ (erefsets \setminus \{ref\}))) \\
\qquad\ \ \vee erefsets = \varnothing \wedge DistExprRefJoin\ erefsets = \varnothing
\end{array}$$

$$\begin{array}{|l}
EnvJoin: Env \times Env \pfun Env \\
\hline
\forall env_1, env_2: Env \\
\quad \bullet\ \exists rel_1, rel_2: ExprShareRelation; ref_1, ref_2: ExprRefSet \\
\qquad \mid\ (rel_1, ref_1) = env_1 \wedge (rel_2, ref_2) = env_2 \\
\qquad \bullet\ EnvJoin\ (env_1, env_2) = rel_1 \cup rel_2 \mapsto ExprRefJoin\ (ref_1, ref_2)
\end{array}$$

$$\begin{array}{|l}
DistEnvJoin: \mathbb{F}\ Env \pfun Env \\
\hline
\forall envs: \mathbb{F}\ Env \\
\quad \bullet\ envs = \varnothing \\
\quad\ \ \wedge (\exists env: envs \\
\qquad\quad \bullet\ DistEnvJoin\ envs = EnvJoin\ (env, (DistEnvJoin\ (envs \setminus \{env\})))) \\
\quad\ \ \vee envs = \varnothing \wedge DistEnvJoin\ envs = (\varnothing, \varnothing)
\end{array}$$

$$\begin{array}{|l}
GetStaticVars: SCJmSafeProgram \pfun \mathbb{P}\ LExpr \\
\hline
\forall p: SCJmSafeProgram \bullet\ GetStaticVars\ p = \{\ d: p.static \bullet\ ID\ (var\ d.var)\ \}
\end{array}$$

$$\begin{array}{|l}
RaiseRC: RefCon \pfun RefCon \\
\hline
\forall rc: RefCon \\
\quad \bullet\ \exists h: Name; n: \mathbb{N} \\
\qquad \bullet\ rc = MMem \wedge RaiseRC\ rc = IMem \\
\qquad\ \ \vee rc = PRMem\ h \wedge RaiseRC\ rc = MMem \\
\qquad\ \ \vee rc = TPMem\ (h, n) \wedge n > 0 \wedge RaiseRC\ rc = TPMem\ (h, (n - 1)) \\
\qquad\ \ \vee rc = TPMem\ (h, n) \wedge n = 0 \wedge RaiseRC\ rc = PRMem\ h \\
\qquad\ \ \vee rc = TPMMem\ n \wedge n > 0 \wedge RaiseRC\ rc = TPMMem\ (n - 1) \\
\qquad\ \ \vee rc = TPMMem\ n \wedge n = 0 \wedge RaiseRC\ rc = MMem
\end{array}$$

$$\begin{array}{|l}
RaiseRCBy: RefCon \times \mathbb{N} \pfun RefCon \\
\hline
\forall rc: RefCon; count: \mathbb{N} \\
\quad \bullet\ count > 0 \\
\quad\ \ \wedge RaiseRCBy\ (rc, count) = RaiseRCBy\ ((RaiseRC\ rc), (count - 1)) \\
\quad\ \ \vee count = 0 \wedge RaiseRCBy\ (rc, count) = rc
\end{array}$$

$$\begin{array}{|l}
\mathit{LowerRC}: \mathit{RefCon} \rightarrowtail \mathit{RefCon} \\
\hline
\forall\ \mathit{rc}: \mathit{RefCon} \\
\quad \bullet\ \exists\ h: \mathit{Name};\ n: \mathbb{N} \\
\qquad \bullet\ \mathit{rc} = \mathit{MMem} \land \mathit{LowerRC}\ \mathit{rc} = \mathit{TPMMem}\ 0 \\
\qquad\ \lor \mathit{rc} = \mathit{PRMem}\ h \land \mathit{LowerRC}\ \mathit{rc} = \mathit{TPMem}\ (h, 0) \\
\qquad\ \lor \mathit{rc} = \mathit{TPMem}\ (h, n) \land \mathit{LowerRC}\ \mathit{rc} = \mathit{TPMem}\ (h, (n + 1)) \\
\qquad\ \lor \mathit{rc} = \mathit{TPMMem}\ n \land \mathit{LowerRC}\ \mathit{rc} = \mathit{TPMMem}\ (n + 1)
\end{array}$$

$$\begin{array}{|l}
\mathit{LowerRCBy}: \mathit{RefCon} \times \mathbb{N} \rightarrowtail \mathit{RefCon} \\
\hline
\forall\ \mathit{rc}: \mathit{RefCon};\ \mathit{count}: \mathbb{N} \\
\quad \bullet\ \mathit{count} > 0 \\
\quad\ \land \mathit{LowerRCBy}\ (\mathit{rc}, \mathit{count}) = \mathit{LowerRCBy}\ ((\mathit{LowerRC}\ \mathit{rc}), (\mathit{count} - 1)) \\
\quad\ \lor \mathit{count} = 0 \land \mathit{LowerRCBy}\ (\mathit{rc}, \mathit{count}) = \mathit{rc}
\end{array}$$

$$\begin{array}{|l}
\mathit{RCsFromMRC}: \mathit{MetaRefCon} \times \mathit{RefCon} \times \mathit{ExprRefSet} \times \mathit{LExpr} \rightarrowtail \mathbb{P}\ \mathit{RefCon} \\
\hline
\forall\ \mathit{mrc}: \mathit{MetaRefCon};\ \mathit{rc}: \mathit{RefCon};\ \mathit{refSet}: \mathit{ExprRefSet};\ \mathit{cexpr}: \mathit{LExpr} \\
\quad \bullet\ \mathit{mrc} = \mathit{Current} \land \mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{refSet}, \mathit{cexpr}) = \{\mathit{rc}\} \\
\quad\ \lor (\exists\ n: \mathbb{N} \mid \mathit{mrc} = \mathit{CurrentPrivate}\ n \\
\qquad \bullet\ \mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{refSet}, \mathit{cexpr}) = \{\mathit{LowerRCBy}\ (\mathit{rc}, n)\}) \\
\quad\ \lor (\exists\ n: \mathbb{N} \mid \mathit{mrc} = \mathit{CurrentPlus}\ n \\
\qquad \bullet\ \mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{refSet}, \mathit{cexpr}) = \{\mathit{RaiseRCBy}\ (\mathit{rc}, n)\}) \\
\quad\ \lor (\exists\ \mathit{rcs}: \mathbb{P}\ \mathit{RefCon} \mid \mathit{mrc} = \mathit{Rcs}\ \mathit{rcs} \\
\qquad \bullet\ \mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{refSet}, \mathit{cexpr}) = \mathit{rcs}) \\
\quad\ \lor (\exists\ e: \mathit{LExpr} \mid \mathit{mrc} = \mathit{Erc}\ e \land e = \mathit{This} \\
\qquad \bullet\ \mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{refSet}, \mathit{cexpr}) = \mathit{refSet}\ \mathit{cexpr}) \\
\quad\ \lor (\exists\ e: \mathit{LExpr} \mid \mathit{mrc} = \mathit{Erc}\ e \land e \neq \mathit{This} \\
\qquad \bullet\ \mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{refSet}, \mathit{cexpr}) = \mathit{refSet}\ e)
\end{array}$$

$$\begin{array}{|l|}
\hline
[X] \\
\hline
\mathit{SeqRestriction}: \mathrm{seq}\ X \times \mathbb{P}\ \mathbb{N} \longrightarrow \mathrm{seq}\ X \\
\hline
\forall\ s: \mathrm{seq}\ X;\ n: \mathbb{P}\ \mathbb{N} \bullet\ \mathit{SeqRestriction}\ (s, n) = \mathit{squash}\ (n \lhd s) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\mathit{getFirstExpr}: \mathit{LExpr} \rightarrowtail \mathit{LExpr} \\
\hline
\forall\ e: \mathit{LExpr} \\
\quad \bullet\ \exists\ \mathit{fa}: \mathit{FieldAccess} \\
\qquad \bullet\ e = \mathit{FA}\ \mathit{fa} \land \mathit{getFirstExpr}\ e = \mathit{ID}\ (\mathit{head}\ \mathit{fa}) \\
\qquad\ \lor e \neq \mathit{FA}\ \mathit{fa} \land \mathit{getFirstExpr}\ e = e
\end{array}$$

$$\begin{array}{|l}
\mathit{getLastExpr}: \mathit{LExpr} \rightarrowtail \mathit{LExpr} \\
\hline
\forall\ e: \mathit{LExpr} \\
\quad \bullet\ \exists\ \mathit{fa}: \mathit{FieldAccess} \\
\qquad \bullet\ e = \mathit{FA}\ \mathit{fa} \land \mathit{getLastExpr}\ e = \mathit{ID}\ (\mathit{last}\ \mathit{fa}) \\
\qquad\ \lor e \neq \mathit{FA}\ \mathit{fa} \land \mathit{getLastExpr}\ e = e
\end{array}$$

$$
\begin{array}{|l}
\mathit{getFrontOfExpr}: \mathit{LExpr} \nrightarrow \mathit{LExpr} \\
\hline
\forall\, e: \mathit{LExpr} \\
\quad \bullet\ \exists\, \mathit{fa}: \mathit{FieldAccess} \\
\qquad \bullet\ e = \mathit{FA}\ \mathit{fa} \land \mathit{getFrontOfExpr}\ e = \mathit{FA}\ (\mathit{front}\ \mathit{fa}) \\
\qquad\ \ \lor e \neq \mathit{FA}\ \mathit{fa} \land \mathit{getFrontOfExpr}\ e = e
\end{array}
$$

$$
\begin{array}{|l}
\mathit{MergeShareExprExprs}: \mathit{LExpr} \times \mathit{LExpr} \times \mathit{LExpr} \nrightarrow \mathit{LExpr} \\
\hline
\forall\, \mathit{newle}, \mathit{newre}, \mathit{sharee}: \mathit{LExpr} \\
\quad \bullet\ (\exists\, \mathit{id}: \mathit{Identifier} \\
\qquad \mid\ \mathit{newle} = \mathit{Val} \lor \mathit{newre} = \mathit{Val} \lor \mathit{sharee} = \mathit{Val} \lor \mathit{sharee} = \mathit{ID}\ \mathit{id} \\
\qquad \bullet\ \mathit{MergeShareExprExprs}\ (\mathit{newle}, \mathit{newre}, \mathit{sharee}) = \mathit{newle}) \\
\quad\ \lor (\exists\, \mathit{fa}_1, \mathit{fa}_2: \mathit{FieldAccess} \mid\ \mathit{sharee} = \mathit{FA}\ \mathit{fa}_1 \land \mathit{newre} = \mathit{FA}\ \mathit{fa}_2 \\
\qquad \bullet\ ((\exists\, \mathit{id}: \mathit{Identifier} \mid\ \mathit{newle} = \mathit{ID}\ \mathit{id} \\
\qquad\qquad \bullet\ \mathit{MergeShareExprExprs}\ (\mathit{newle}, \mathit{newre}, \mathit{sharee}) \\
\qquad\qquad\quad = \mathit{FA}\ (\langle \mathit{id} \rangle \\
\qquad\qquad\qquad\quad {}^\frown\ \mathit{SeqRestriction}\ (\mathit{fa}_1, (\#\, \mathit{fa}_2 + 1 \mathrel{..} \#\, \mathit{fa}_1)))) \\
\qquad\quad \lor (\exists\, \mathit{fa}_3: \mathit{FieldAccess} \mid\ \mathit{newle} = \mathit{FA}\ \mathit{fa}_3 \\
\qquad\qquad \bullet\ \mathit{MergeShareExprExprs}\ (\mathit{newle}, \mathit{newre}, \mathit{sharee}) \\
\qquad\qquad\quad = \mathit{FA}\ (\mathit{fa}_3 \\
\qquad\qquad\qquad\quad {}^\frown\ \mathit{SeqRestriction}\ (\mathit{fa}_1, \\
\qquad\qquad\qquad\qquad\qquad (\#\, \mathit{fa}_2 + 1 \mathrel{..} \#\, \mathit{fa}_1)))))) \\
\quad\ \lor (\exists\, \mathit{fa}_1: \mathit{FieldAccess}; \mathit{id}_1: \mathit{Identifier} \\
\qquad \mid\ \mathit{sharee} = \mathit{FA}\ \mathit{fa}_1 \land \mathit{newre} = \mathit{ID}\ \mathit{id}_1 \\
\qquad \bullet\ ((\exists\, \mathit{id}_2: \mathit{Identifier} \mid\ \mathit{newle} = \mathit{ID}\ \mathit{id}_2 \\
\qquad\qquad \bullet\ \mathit{MergeShareExprExprs}\ (\mathit{newle}, \mathit{newre}, \mathit{sharee}) \\
\qquad\qquad\quad = \mathit{FA}\ (\langle \mathit{id}_2 \rangle\ {}^\frown\ \mathit{tail}\ \mathit{fa}_1)) \\
\qquad\quad \lor (\exists\, \mathit{fa}_2: \mathit{FieldAccess} \mid\ \mathit{newle} = \mathit{FA}\ \mathit{fa}_2 \\
\qquad\qquad \bullet\ \mathit{MergeShareExprExprs}\ (\mathit{newle}, \mathit{newre}, \mathit{sharee}) \\
\qquad\qquad\quad = \mathit{FA}\ (\mathit{fa}_2\ {}^\frown\ \mathit{tail}\ \mathit{fa}_1))))
\end{array}
$$

$$
\begin{array}{|l}
\mathit{MergeExprs}: \mathit{Expr} \times \mathit{Expr} \times \mathit{SCJmSafeProgram} \nrightarrow \mathit{LExpr} \\
\hline
\forall\, e_1, e_2: \mathit{Expr}; p: \mathit{SCJmSafeProgram} \\
\quad \bullet\ e_2 = \mathit{Null} \land \mathit{MergeExprs}\ (e_1, e_2, p) = e_1 \\
\quad\ \lor (e_1 = \mathit{Null} \lor e_1 = \mathit{This} \lor e_1 = \mathit{Val}) \land \mathit{MergeExprs}\ (e_1, e_2, p) = e_2 \\
\quad\ \lor e_2 \in \mathit{GetStaticVars}\ p \land \mathit{MergeExprs}\ (e_1, e_2, p) = e_2 \\
\quad\ \lor (\exists\, \mathit{fa}_1, \mathit{fa}_2: \mathit{FieldAccess}; \mathit{ae}: \mathit{ArrayElement}; v: \mathit{Variable} \\
\qquad \bullet\ (e_1 = \mathit{FA}\ \mathit{fa}_1 \\
\qquad\quad \land e_2 = \mathit{FA}\ \mathit{fa}_2 \\
\qquad\quad \land \mathit{MergeExprs}\ (e_1, e_2, p) = \mathit{FA}\ (\mathit{fa}_1\ {}^\frown\ \mathit{fa}_2) \\
\qquad\quad \lor e_1 = \mathit{FA}\ \mathit{fa}_1 \\
\qquad\quad\ \ \land e_2 = \mathit{ID}\ (\mathit{arrayElement}\ \mathit{ae}) \\
\qquad\quad\ \ \land \mathit{MergeExprs}\ (e_1, e_2, p) = \mathit{FA}\ (\mathit{fa}_1\ {}^\frown\ \langle \mathit{arrayElement}\ \mathit{ae} \rangle) \\
\qquad\quad \lor e_1 = \mathit{FA}\ \mathit{fa}_1 \\
\qquad\quad\ \ \land e_2 = \mathit{ID}\ (\mathit{var}\ v) \\
\qquad\quad\ \ \land \mathit{MergeExprs}\ (e_1, e_2, p) = \mathit{FA}\ (\mathit{fa}_1\ {}^\frown\ \langle \mathit{var}\ v \rangle))) \\
\quad\ \lor (\exists\, \mathit{fa}_1: \mathit{FieldAccess}; \mathit{ae}: \mathit{ArrayElement}; v_1, v_2: \mathit{Variable} \\
\qquad \bullet\ (e_1 = \mathit{ID}\ (\mathit{var}\ v_1) \\
\qquad\quad \land e_2 = \mathit{FA}\ \mathit{fa}_1 \\
\qquad\quad \land \mathit{MergeExprs}\ (e_1, e_2, p) = \mathit{FA}\ (\langle \mathit{var}\ v_1 \rangle\ {}^\frown\ \mathit{fa}_1) \\
\qquad\quad \lor e_1 = \mathit{ID}\ (\mathit{var}\ v_1) \\
\qquad\quad\ \ \land e_2 = \mathit{ID}\ (\mathit{arrayElement}\ \mathit{ae})
\end{array}
$$

```
          $\land MergeExprs\ (e_1, e_2, p) = FA\ (\langle var\ v_1\rangle \frown \langle arrayElement\ ae\rangle)$
        $\lor e_1 = ID\ (var\ v_1)$
          $\land e_2 = ID\ (var\ v_2)$
          $\land MergeExprs\ (e_1, e_2, p) = FA\ (\langle var\ v_1\rangle \frown \langle var\ v_2\rangle)))$
  $\lor (\exists fa_1: FieldAccess; ae_1, ae_2: ArrayElement; v: Variable$
      $\bullet\ (e_1 = ID\ (arrayElement\ ae_1)$
        $\land e_2 = FA\ fa_1$
        $\land MergeExprs\ (e_1, e_2, p) = FA\ (\langle arrayElement\ ae_1\rangle \frown fa_1)$
        $\lor e_1 = ID\ (arrayElement\ ae_1)$
          $\land e_2 = ID\ (arrayElement\ ae_2)$
          $\land MergeExprs\ (e_1, e_2, p)$
             $= FA\ (\langle arrayElement\ ae_1\rangle \frown \langle arrayElement\ ae_2\rangle)$
        $\lor e_1 = ID\ (arrayElement\ ae_1)$
          $\land e_2 = ID\ (var\ v)$
          $\land MergeExprs\ (e_1, e_2, p)$
             $= FA\ (\langle arrayElement\ ae_1\rangle \frown \langle var\ v\rangle)))$
```

```
$GetDecRefCon: Dec \rightarrow\!\!\!\!\!+\ \mathbb{P}\ RefCon$
-----
$\forall d: Dec$
  $\bullet\ d.var.varType.isPrimitive = True \land GetDecRefCon\ d = \{Prim\}$
   $\lor d.var.varType.isPrimitive = False \land GetDecRefCon\ d = \{\}$
```

```
$AddDecToEnv: Env \times Dec \rightarrow\!\!\!\!\!+\ Env$
-----
$\forall env: Env; d: Dec$
  $\bullet\ \exists rel: ExprShareRelation; ref: ExprRefSet \mid env = (rel, ref)$
     $\bullet\ AddDecToEnv\ (env, d)$
        $= ExprShareAdd\ ((ID\ (var\ d.var)), (ID\ (var\ d.var)), rel)$
          $\mapsto ExprRefAdd\ ((ID\ (var\ d.var)), (GetDecRefCon\ d), ref)$
```

```
$AddDecsToEnv: Env \times \mathbb{P}\ Dec \rightarrow\!\!\!\!\!+\ Env$
-----
$\forall env: Env; decs: \mathbb{P}\ Dec$
  $\bullet\ (\exists d: decs$
       $\bullet\ AddDecsToEnv\ (env, decs)$
          $= AddDecsToEnv\ ((AddDecToEnv\ (env, d)), (decs \setminus \{d\})))$
   $\lor decs = \varnothing \land AddDecsToEnv\ (env, decs) = env$
```

```
$GetExprType: LExpr \rightarrow\!\!\!\!\!+\ Name$
-----
$\forall e: LExpr$
  $\bullet\ (\exists v: Variable \mid e = ID\ (var\ v) \bullet GetExprType\ e = v.varType.type)$
   $\lor (\exists ae: ArrayElement \mid e = ID\ (arrayElement\ ae)$
       $\bullet\ GetExprType\ e = ae.type)$
   $\lor (\exists fa: FieldAccess \mid e = FA\ fa$
       $\bullet\ GetExprType\ e = GetExprType\ (ID\ (last\ fa)))$
```

```
$MatchingTypes: \mathrm{seq}\ Expr \times \mathrm{seq}\ Variable \rightarrow\!\!\!\!\!+\ Boolean$
-----
$\forall args: \mathrm{seq}\ Expr; params: \mathrm{seq}\ Variable$
```

```
  • let matchSet ==
        { $n: 1 \mathrel{..} \# args$
            | $GetExprType\ (args\ n) = (params\ n).varType.type \bullet False$ }
    • $\# args \neq \# params \land MatchingTypes\ (args, params) = False$
      $\lor \# args = \# params$
       $\land matchSet = \emptyset$
       $\land MatchingTypes\ (args, params) = True$
      $\lor \# args = \# params$
       $\land False \in matchSet$
       $\land MatchingTypes\ (args, params) = False$
```

```
$MatchingTypesMethSig: \mathrm{seq}\ Name \times \mathrm{seq}\ Variable \pfun Boolean$
────────────
$\forall types: \mathrm{seq}\ Name; params: \mathrm{seq}\ Variable$
  • let matchSet ==
        { $n: 1 \mathrel{..} \# types \mid types\ n = (params\ n).varType.type \bullet False$ }
    • $\# types \neq \# params \land MatchingTypesMethSig\ (types, params) = False$
      $\lor \# types = \# params$
       $\land matchSet = \emptyset$
       $\land MatchingTypesMethSig\ (types, params) = True$
      $\lor \# types = \# params$
       $\land False \in matchSet$
       $\land MatchingTypesMethSig\ (types, params) = False$
```

```
$GetMethodsFromSigs: \mathbb{P}\ MethodSig \times SCJmSafeProgram \pfun \mathbb{P}\ Method$
────────────
$\forall sigs: \mathbb{P}\ MethodSig; p: SCJmSafeProgram$
  • let methods == $p.safelet.methods \cup p.missionSeq.methods$
            $\cup \bigcup \{ mission: p.missions \bullet mission.methods \}$
            $\cup \bigcup \{ handler: p.handlers \bullet handler.methods \}$
            $\cup \bigcup \{ class: p.classes \bullet class.methods \}$
    • $GetMethodsFromSigs\ (sigs, p)$
      $= \bigcup \{ sig: sigs$
          $\bullet \{ m: methods$
              $\mid m.name = sig.name$
               $\land sig.returnType = m.type$
               $\land MatchingTypesMethSig\ (sig.paramTypes,$
                          $m.params) = True \bullet m \} \}$
```

```
$GetConstr: Name \times \mathrm{seq}\ Expr \times SCJmSafeProgram \pfun Method$
────────────
$\forall name: Name; args: \mathrm{seq}\ Expr; p: SCJmSafeProgram$
  • let constrs == $p.safelet.constrs \cup p.missionSeq.constrs$
            $\cup \bigcup \{ mission: p.missions \bullet mission.constrs \}$
            $\cup \bigcup \{ handler: p.handlers \bullet handler.constrs \}$
            $\cup \bigcup \{ class: p.classes \bullet class.constrs \}$
    • $\exists m: Method \mid m.name = name \land MatchingTypes\ (args, m.params) = True$
        • $GetConstr\ (name, args, p) = m$
```

```
$AddAsgnFields: LExpr \times Expr \times Env \pfun Env$
────────────
$\forall env: Env; le: LExpr; re: Expr$
```

```
  • ∃ rel: ExprShareRelation; ref: ExprRefSet | env = (rel, ref)
     • AddAsgnFields (le, re, env)
        = ExprShareAddSet ({ e1, e2: LExpr
                             | (e1, e2) ∈ rel
                               ∧ PrefixOf (re, e1) = True
                             • ((MergeShareExprExprs (le, re, e1)),
                               e1) }, rel)
          ↦ ExprRefUpdateSet ({ e1, e2: LExpr
                               | (e1, e2) ∈ rel
                                 ∧ PrefixOf (re, e1) = True
                               • ((MergeShareExprExprs (le, re,
                                                          e1)),
                                 (ref e1)) }, ref)
```

```
AddAsgnFieldsSet: ℙ (LExpr × Expr) × Env ⇸ Env
──────────────────────────────
∀ shareSet: ℙ (LExpr × LExpr); env: Env
  • ∃ rel: ExprShareRelation; ref: ExprRefSet | env = (rel, ref)
     • AddAsgnFieldsSet (shareSet, env)
        = ∪ { le: LExpr; re: Expr | (le, re) ∈ shareSet
               • (ExprShareAddSet ({ e1, e2: LExpr
                                     | (e1, e2) ∈ rel
                                       ∧ PrefixOf (re, e1) = True
                                     • ((MergeShareExprExprs (le,
                                                                re,
                                                                e1)),
                                       e1) }, rel)) }
          ↦ DistExprRefJoin { le: LExpr; re: Expr | (le, re) ∈ shareSet
                               • (ExprRefUpdateSet ({ e1, e2: LExpr
                                                      | (e1, e2)
                                                        ∈ rel
                                                        ∧ PrefixOf (re,
                                                                    e1)
                                                          = True
                                                      • ((MergeShareExprExprs (le,
                                                                                 re,
                                                                                 e1)),
                                                        (ref e1)) },
                                                  ref)) }
```

```
AddAsgnFieldsShareSet: ℙ (LExpr × Expr) × ExprShareRelation
                  ⇸ ExprShareRelation
──────────────────────────────
∀ shareSet: ℙ (LExpr × LExpr); rel: ExprShareRelation
  • AddAsgnFieldsShareSet (shareSet, rel)
     = ∪ { le: LExpr; re: Expr | (le, re) ∈ shareSet
            • (ExprShareAddSet ({ e1, e2: LExpr
                                  | (e1, e2) ∈ rel
                                    ∧ PrefixOf (re, e1) = True
                                  • ((MergeShareExprExprs (le, re,
                                                             e1)),
                                    e1) }, rel)) }
```

```
AddAsgnFieldsRefSet: ℙ (LExpr × Expr) × ExprShareRelation × ExprRefSet
                 ⇸ ExprRefSet
─────────────────────────────
∀ shareSet: ℙ (LExpr × LExpr); rel: ExprShareRelation; ref: ExprRefSet
  • AddAsgnFieldsRefSet (shareSet, rel, ref)
    = DistExprRefJoin {  le: LExpr; re: Expr | (le, re) ∈ shareSet
                        • (ExprRefUpdateSet ({  e1, e2: LExpr
                                               | (e1, e2) ∈ rel
                                                 ∧ PrefixOf (re, e1)
                                                    = True
                                               • ((MergeShareExprExprs (le,
                                                                        re,
                                                                        e1)),
                                                 (ref e1))  },
                                            ref))  }
```

```
UpdateEqualExprs: LExpr × Expr × Env ⇸ Env
─────────────────────────────
∀ env: Env; le: LExpr; re: Expr
  • ∃ rel: ExprShareRelation; ref: ExprRefSet | env = (rel, ref)
    • (∃ fa: FieldAccess | le = FA fa
        • (let shareSet ==
                 {  e1, e2, equale: LExpr
                    | (e1, e2) ∈ rel
                      ∧ FA (front fa) = e1
                      ∧ equale = MergeShareExprExprs (e2, e1, le)
                    • (equale, re)  };
               refSet ==
                 {  e1, e2, equale: LExpr
                    | (e1, e2) ∈ rel
                      ∧ FA (front fa) = e1
                      ∧ equale = MergeShareExprExprs (e2, e1, le)
                    • (equale, ref re)  }
           • UpdateEqualExprs (le, re, env)
             = AddAsgnFieldsSet (shareSet,
                                (ExprShareAddSet (shareSet, rel)
                                 ↦ ExprRefUpdateSet (refSet,
                                                     ref)))))
      ∨ (∃ ae: ArrayElement | le = ID (arrayElement ae)
        • (let shareSet ==
                   {  e1, e2, equale: LExpr; v: Variable;
                      fa: FieldAccess
                      | (e1, e2) ∈ rel
                        ∧ v.name = ae.name
                        ∧ last fa = var v
                        ∧ (e1 = ID (var v) ∨ e1 = FA fa)
                        ∧ equale
                           = MergeShareExprExprs (e2, e1, le)
                      • (equale, re)  };
               refSet ==
                   {  e1, e2, equale: LExpr; v: Variable;
                      fa: FieldAccess
                      | (e1, e2) ∈ rel
                        ∧ v.name = ae.name
                        ∧ last fa = var v
                        ∧ (e1 = ID (var v) ∨ e1 = FA fa)
                        ∧ equale
                           = MergeShareExprExprs (e2, e1, le)
                      • (equale, ref re) }
              • UpdateEqualExprs (le, re, env)
                 = AddAsgnFieldsSet (shareSet,
                         (ExprShareAddSet (shareSet,
                                           rel)
                          ↦ ExprRefUpdateSet (refSet,
                                              ref)))))
```

```
UpdateEqualExprsSet: ℙ (LExpr × Expr) × Env ⇸ Env
────────────────────────────
∀ env: Env; set: ℙ (LExpr × Expr)
  • ∃ rel: ExprShareRelation; ref: ExprRefSet | env = (rel, ref)
     • UpdateEqualExprsSet (set, env)
        = ∪ {  le: LExpr; re: Expr | (le, re) ∈ set
           • (∪ {  fa: FieldAccess | le = FA fa
                 • (let shareSet ==
                        {  e1, e2, equale: LExpr
                           | (e1, e2) ∈ rel
                             ∧ FA (front fa) = e1
                             ∧ equale
                                = MergeShareExprExprs (e2,
                                                      e1,
                                                      le)
                           • (equale, re) }
                    • (AddAsgnFieldsShareSet (shareSet,
                                     (ExprShareAddSet (shareSet,
                                                       rel))))) }
              ∪ ∪ {  ae: ArrayElement | le = ID (arrayElement ae)
                  • (let shareSet ==
                         {  e1, e2, equale: LExpr;
                            v: Variable; fa: FieldAccess
                            | (e1, e2) ∈ rel
                              ∧ v.name = ae.name
                              ∧ last fa = var v
                              ∧ (e1 = ID (var v)
                                ∨ e1 = FA fa)
                              ∧ equale
                                 = MergeShareExprExprs (e2,
                                                       e1,
                                                       le)
                            • (equale, re) }
                     • (AddAsgnFieldsShareSet (shareSet,
                                      (ExprShareAddSet (shareSet,
                                                        rel))))) }) }
          ↦ DistExprRefJoin {  le: LExpr; re: Expr | (le, re) ∈ set
                     • (DistExprRefJoin ({  fa: FieldAccess
                                          | le = FA fa
                                          • (let shareSet ==
                                                 {  e1,
                                                    e2,
                                                    equale: LExpr
                                                     | (e1,
                                                        e2)
                                                        ∈ rel
                                                      ∧ FA (front fa)
                                                          = e1
                                                      ∧ equale
                                                          = MergeShareExprExprs (e2,
                                                                                 e1,
                                                                                 le)
                                                    • (equale,
                                                       re) };
                                             refSet ==
                                                 {  e1,
                                                    e2,
                                                    equale: LExpr
                                                      | (e1,
                                                        e2)
                                                        ∈ rel
                                                      ∧ FA (front fa)
                                                          = e1
                                                      ∧ equale
                                                          = MergeShareExprExprs (e2,
                                                                                 e1,
                                                                                 le)
                                                    • (equale,
                                                       ref re) }
                                            • AddAsgnFieldsRefSet (shareSet,
                                                                   rel,
                                                                   (ExprRefUpdateSet (refSet,
                                                                                      ref)))) }
                                  ∪ {  ae: ArrayElement
                                        | le
                                            = ID (arrayElement ae)
                                      • (let shareSet ==
                                                {  e1,
                                                   e2,
                                                   equale: LExpr;
                                                   v: Variable;
                                                   fa: FieldAccess
                                                     | (e1,
                                                       e2)
                                                       ∈ rel
                                                     ∧ v.name
                                                         = ae.name
                                                     ∧ last fa
                                                         = var v
                                                     ∧ (e1
                                                          = ID (var v)
                                                        ∨ e1
                                                            = FA fa)
                                                     ∧ equale
                                                         = MergeShareExprExprs (e2,
                                                                                e1,
                                                                                le)
                                                   • (equale,
                                                      re) };
                                            refSet ==
                                          {  e1,
                                             e2,
                                             equale: LExpr;
                                             v: Variable;
                                             fa: FieldAccess
                                               | (e1,
                                                  e2)
                                                  ∈ rel
                                                ∧ v.name
                                                   = ae.name
                                                ∧ last fa
                                                   = var v
                                                ∧ (e1
                                                     = ID (var v)
                                                  ∨ e1
                                                     = FA fa)
                                                ∧ equale
                                                   = MergeShareExprExprs (e2,
                                                                          e1,
                                                                          le)
                                               • (equale,
                                                  ref re)  }
                                      • AddAsgnFieldsRefSet (shareSet,
                                                             rel,
                                                             (ExprRefUpdateSet (refSet,
                                                                                ref))))  }))  }
```

```
ExprRefAddEnvAsgn: LExpr × Expr × Env ⇸ Env
───────────────────────────
∀ env: Env; le: LExpr; re: Expr
  • ∃ rel: ExprShareRelation; ref: ExprRefSet |  env = (rel, ref)
      • ExprRefAddEnvAsgn (le, re, env)
          = rel ↦ ExprRefUpdate (le, (ref re), ref)
```

```
CalcEnvAssignment: Env × LExpr × Expr × LExpr × RefCon × SCJmSafeProgram ⇸ Env
───────────────────────────
∀ env: Env; le, cexpr: LExpr; re: Expr; rc: RefCon; p: SCJmSafeProgram
  • re = Val ∧ CalcEnvAssignment (env, le, re, cexpr, rc, p) = env
    ∨ re ≠ Val
     ∧ (∃ newle: LExpr; newre: Expr
          |  newle = MergeExprs (cexpr, le, p)
            ∧ (re = This ∧ newre = cexpr
              ∨ re ≠ This ∧ newre = MergeExprs (cexpr, re, p))
          • ((∃ v: Variable |  le = ID (var v)
                • CalcEnvAssignment (env, le, re, cexpr, rc, p)
                   = AddAsgnFields (newle, newre,
                                    (ExprRefAddEnvAsgn (newle, newre,
                                                        (ExprShareAddEnv (newle,
                                                                          newre,
                                                                          env))))))
            ∨ (∃ fa: FieldAccess; ae: ArrayElement
                 |  le = FA fa ∨ le = ID (arrayElement ae)
                 • CalcEnvAssignment (env, le, re, cexpr, rc, p)
                    = UpdateEqualExprs (newle, newre,
```

$$\begin{array}{l}
\qquad\qquad\qquad\qquad (\mathit{AddAsgnFields}\ (\mathit{newle}, \mathit{newre},\\
\qquad\qquad\qquad\qquad\qquad (\mathit{ExprRefAddEnvAsgn}\ (\mathit{newle},\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{newre},\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{ExprShareAddEnv}\ (\mathit{newle},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{newre},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{env}))))))))))
\end{array}$$

$$\begin{array}{l}
\mathit{UpdateMethodPropertiesCExprShare}: \mathit{ExprShareRelation} \times \mathrm{seq}\ \mathit{Expr} \times \mathbb{P}\ \mathit{LExpr} \times\\
\qquad\qquad \mathit{LExpr} \times \mathit{LExpr} \times \mathit{SCJmSafeProgram}\\
\qquad\qquad \nrightarrow \mathit{ExprShareRelation}\\
\hline
\forall\ \mathit{rel}: \mathit{ExprShareRelation}; \mathit{args}: \mathrm{seq}\ \mathit{Expr}; \mathit{fields}: \mathbb{P}\ \mathit{LExpr}; \mathit{cexpr},\\
\quad \mathit{lexpr}: \mathit{LExpr}; p: \mathit{SCJmSafeProgram}\\
\quad \bullet\ \mathit{UpdateMethodPropertiesCExprShare}\ (\mathit{rel}, \mathit{args}, \mathit{fields}, \mathit{cexpr}, \mathit{lexpr}, p)\\
\qquad = \{\ e1, e2: \mathit{LExpr}\\
\qquad\quad \mid (e1, e2) \in \mathit{rel}\\
\qquad\quad \wedge (\forall\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e1) = \mathit{False})\\
\qquad\quad \wedge (\forall\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e1 \neq \mathit{field})\\
\qquad\quad \wedge (\forall\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e2) = \mathit{False})\\
\qquad\quad \wedge (\forall\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e2 \neq \mathit{field})\\
\qquad\quad \bullet\ (\mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, \mathit{lexpr}, p)), e1, p)\\
\qquad\quad\ \mapsto \mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, \mathit{lexpr}, p)), e2, p))\ \}\\
\qquad \cup \{\ e1, e2: \mathit{LExpr}\\
\qquad\quad \mid (e1, e2) \in \mathit{rel}\\
\qquad\quad \wedge ((\exists\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e1) = \mathit{True})\\
\qquad\qquad \vee (\exists\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e1 = \mathit{field}))\\
\qquad\quad \wedge (\forall\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e2) = \mathit{False})\\
\qquad\quad \wedge (\forall\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e2 \neq \mathit{field})\\
\qquad\quad \bullet\ (\mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, (\mathit{getFrontOfExpr}\ \mathit{lexpr}),\\
\qquad\qquad\qquad\qquad\qquad p)), e1, p)\\
\qquad\quad\ \mapsto \mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, \mathit{lexpr}, p)), e2, p))\ \}\\
\qquad \cup \{\ e1, e2: \mathit{LExpr}\\
\qquad\quad \mid (e1, e2) \in \mathit{rel}\\
\qquad\quad \wedge (\forall\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e1) = \mathit{False})\\
\qquad\quad \wedge (\forall\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e1 \neq \mathit{field})\\
\qquad\quad \wedge ((\exists\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e2) = \mathit{True})\\
\qquad\qquad \vee (\exists\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e2 = \mathit{field}))\\
\qquad\quad \bullet\ (\mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, \mathit{lexpr}, p)), e1, p)\\
\qquad\quad\ \mapsto \mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, (\mathit{getFrontOfExpr}\ \mathit{lexpr}),\\
\qquad\qquad\qquad\qquad\qquad p)), e2, p))\ \}\\
\qquad \cup \{\ e1, e2: \mathit{LExpr}\\
\qquad\quad \mid (e1, e2) \in \mathit{rel}\\
\qquad\quad \wedge ((\exists\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e1) = \mathit{True})\\
\qquad\qquad \vee (\exists\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e1 = \mathit{field}))\\
\qquad\quad \wedge ((\exists\ \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet\ \mathit{PrefixOf}\ (\mathit{arg}, e1) = \mathit{True})\\
\qquad\qquad \vee (\exists\ \mathit{field}: \mathit{fields} \bullet\ \mathit{getFirstExpr}\ e1 = \mathit{field}))\\
\qquad\quad \bullet\ (\mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, (\mathit{getFrontOfExpr}\ \mathit{lexpr}),\\
\qquad\qquad\qquad\qquad\qquad p)), e1, p)\\
\qquad\quad\ \mapsto \mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, (\mathit{getFrontOfExpr}\ \mathit{lexpr}),\\
\qquad\qquad\qquad\qquad\qquad p)), e2, p))\ \}
\end{array}$$

$$\begin{array}{l}
\mathit{UpdateMethodPropertiesCExprRef}: \mathit{MethodRefSet} \times \mathrm{seq}\ \mathit{Expr} \times \mathbb{P}\ \mathit{LExpr} \times \mathit{LExpr} \times\\
\qquad\qquad \mathit{LExpr} \times \mathit{SCJmSafeProgram}\\
\qquad\qquad \nrightarrow \mathit{MethodRefSet}
\end{array}$$

$$
\begin{array}{l}
\forall \mathit{ref}: \mathit{MethodRefSet}; \mathit{args}: \mathrm{seq}\ \mathit{Expr}; \mathit{fields}: \mathbb{P}\ \mathit{LExpr}; \mathit{cexpr}, \mathit{lexpr}: \mathit{LExpr};\\
\quad p: \mathit{SCJmSafeProgram}\\
\quad\bullet\ \mathit{UpdateMethodPropertiesCExprRef}\,(\mathit{ref}, \mathit{args}, \mathit{fields}, \mathit{cexpr}, \mathit{lexpr}, p)\\
\qquad = \{\ e: \mathit{LExpr}; \mathit{mrcs}: \mathbb{P}\ \mathit{MetaRefCon}\\
\qquad\qquad \mid (e, \mathit{mrcs}) \in \mathit{ref}\\
\qquad\qquad\quad \land (\forall \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet \mathit{PrefixOf}\,(\mathit{arg}, e) = \mathit{False})\\
\qquad\qquad\quad \land (\forall \mathit{field}: \mathit{fields} \bullet \mathit{getFirstExpr}\ e \neq \mathit{field})\\
\qquad\qquad \bullet\ \mathit{MergeExprs}\,((\mathit{MergeExprs}\,(\mathit{cexpr}, \mathit{lexpr}, p)), e, p)\\
\qquad\qquad\quad \mapsto \mathit{mrcs} \setminus \{\ \mathit{mrc}: \mathit{mrcs}; e1: \mathit{LExpr} \mid \mathit{mrc} = \mathit{Erc}\ e1 \bullet \mathit{mrc}\ \}\\
\qquad\qquad\quad\ \cup \{\ \mathit{mrc}: \mathit{mrcs}; e1: \mathit{LExpr}\\
\qquad\qquad\qquad\quad \mid \mathit{mrc} = \mathit{Erc}\ e1\\
\qquad\qquad\qquad\qquad \land (\forall \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet \mathit{PrefixOf}\,(\mathit{arg}, e1) = \mathit{False})\\
\qquad\qquad\qquad\qquad \land (\forall \mathit{field}: \mathit{fields} \bullet \mathit{getFirstExpr}\ e1 \neq \mathit{field})\\
\qquad\qquad\qquad\qquad \land e1 = \mathit{This} \bullet \mathit{Erc}\ (\mathit{MergeExprs}\,(\mathit{cexpr}, \mathit{lexpr}, p))\ \}\\
\qquad\qquad\quad\ \cup \{\ \mathit{mrc}: \mathit{mrcs}; e1: \mathit{LExpr}\\
\qquad\qquad\qquad\quad \mid \mathit{mrc} = \mathit{Erc}\ e1\\
\qquad\qquad\qquad\qquad \land (\forall \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet \mathit{PrefixOf}\,(\mathit{arg}, e1) = \mathit{False})\\
\qquad\qquad\qquad\qquad \land (\forall \mathit{field}: \mathit{fields} \bullet \mathit{getFirstExpr}\ e1 \neq \mathit{field})\\
\qquad\qquad\qquad\qquad \land e1 \neq \mathit{This}\\
\qquad\qquad\qquad\quad \bullet\ \mathit{Erc}\ (\mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr}, \mathit{lexpr}, p)),\\
\qquad\qquad\qquad\qquad\qquad\qquad e1, p))\ \}\\
\qquad\qquad\quad\ \cup \{\ \mathit{mrc}: \mathit{mrcs}; e1: \mathit{LExpr}\\
\qquad\qquad\qquad\quad \mid \mathit{mrc} = \mathit{Erc}\ e1\\
\qquad\qquad\qquad\qquad \land ((\exists \mathit{arg}: \mathrm{ran}\ \mathit{args} \bullet \mathit{PrefixOf}\,(\mathit{arg}, e1) = \mathit{True})\\
\qquad\qquad\qquad\qquad\quad \lor (\exists \mathit{field}: \mathit{fields} \bullet \mathit{getFirstExpr}\ e1 = \mathit{field}))\\
\qquad\qquad\qquad\quad \bullet\ \mathit{Erc}\ (\mathit{MergeExprs}\ ((\mathit{MergeExprs}\ (\mathit{cexpr},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{getFrontOfExpr}\ \mathit{lexpr}),\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p)), e1, p))\ \}\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{UpdateMethodPropertiesCExpr}: \mathit{ExprShareRelation} \times \mathit{MethodRefSet} \times \mathrm{seq}\ \mathit{Expr} \times\\
\qquad\qquad\qquad \mathbb{P}\ \mathit{LExpr} \times \mathit{LExpr} \times \mathit{LExpr} \times \mathit{SCJmSafeProgram}\\
\qquad\qquad\qquad \rightarrow\!\!\!\!\!+\ \mathit{ExprShareRelation} \times \mathit{MethodRefSet}\\
\hline
\forall \mathit{rel}: \mathit{ExprShareRelation}; \mathit{ref}: \mathit{MethodRefSet}; \mathit{args}: \mathrm{seq}\ \mathit{Expr}; \mathit{fields}: \mathbb{P}\ \mathit{LExpr};\\
\quad \mathit{cexpr}, \mathit{lexpr}: \mathit{LExpr}; p: \mathit{SCJmSafeProgram}\\
\quad\bullet\ \mathit{UpdateMethodPropertiesCExpr}\ (\mathit{rel}, \mathit{ref}, \mathit{args}, \mathit{fields}, \mathit{cexpr}, \mathit{lexpr}, p)\\
\qquad = (\mathit{UpdateMethodPropertiesCExprShare}\ (\mathit{rel}, \mathit{args}, \mathit{fields}, \mathit{cexpr}, \mathit{lexpr},\\
\qquad\qquad\qquad\qquad p),\\
\qquad\quad \mathit{UpdateMethodPropertiesCExprRef}\,(\mathit{ref}, \mathit{args}, \mathit{fields}, \mathit{cexpr}, \mathit{lexpr},\\
\qquad\qquad\qquad\qquad p))
\end{array}
$$

$$
\begin{array}{l}
\mathit{RefSetFromMethodRef}: \mathit{MethodRefSet} \times \mathit{ExprRefSet} \times \mathit{RefCon} \times \mathit{LExpr} \rightarrow\!\!\!\!\!+\ \mathit{ExprRefSet}\\
\hline
\forall \mathit{methRef}: \mathit{MethodRefSet}; \mathit{ref}: \mathit{ExprRefSet}; \mathit{rc}: \mathit{RefCon}; \mathit{cexpr}: \mathit{LExpr}\\
\quad\bullet\ \mathit{RefSetFromMethodRef}\,(\mathit{methRef}, \mathit{ref}, \mathit{rc}, \mathit{cexpr})\\
\qquad = \{\ e: \mathit{LExpr}; \mathit{mrcs}: \mathbb{P}\ \mathit{MetaRefCon} \mid (e, \mathit{mrcs}) \in \mathit{methRef}\\
\qquad\qquad \bullet\ e \mapsto \bigcup \{\ \mathit{mrc}: \mathit{mrcs} \bullet (\mathit{RCsFromMRC}\ (\mathit{mrc}, \mathit{rc}, \mathit{ref}, \mathit{cexpr}))\ \}\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ApplyMethodProperties}: \mathit{Method} \times \mathit{MethodProperties} \times \mathit{Env} \times \mathrm{seq}\ \mathit{Expr} \times \mathit{LExpr} \times\\
\qquad\qquad \mathit{LExpr} \times \mathit{RefCon} \times \mathit{SCJmSafeProgram}\\
\qquad\qquad \rightarrow\!\!\!\!\!+\ \mathit{Env}\\
\hline
\end{array}
$$

```
∀ m: Method; properties: MethodProperties; args: seq Expr; env: Env; cexpr,
  lexpr: LExpr; rc: RefCon; p: SCJmSafeProgram
   • ∃ rel, methRel: ExprShareRelation; ref: ExprRefSet;
      methRef: MethodRefSet
      | env = (rel, ref) ∧ properties = (methRel, methRef)
      • let updatedShare ==
              UpdateMethodPropertiesCExprShare (methRel, args,
                                        m.visibleFields, cexpr,
                                        lexpr, p);
          updatedRef ==
              UpdateMethodPropertiesCExprRef (methRef, args,
                                        m.visibleFields, cexpr,
                                        lexpr, p)
        • ApplyMethodProperties (m, properties, env, args, cexpr, lexpr,
                         rc, p)
          = UpdateEqualExprsSet (updatedShare,
                         (AddAsgnFieldsSet (updatedShare,
                                    (ExprShareAddSet (updatedShare,
                                              rel)
                                    ↦ ExprRefUpdateSet ((RefSetFromMethodRef (updatedRef,
                                                                 ref,
                                                                 rc,
                                                                 cexpr)),
                                              ref)))))
```

```
UpdateParamWithArg: LExpr × Variable × LExpr ⇸ LExpr
────────────────────────
∀ comp, arg: LExpr; param: Variable
   • (∃ v: Variable | comp = ID (var v)
        • v = param ∧ UpdateParamWithArg (comp, param, arg) = arg
          ∨ v ≠ param ∧ UpdateParamWithArg (comp, param, arg) = comp)
     ∨ (∃ fa: FieldAccess | comp = FA fa
        • (head fa = var param
          ∧ (∃ fa2: FieldAccess | arg = FA fa2
               • UpdateParamWithArg (comp, param, arg)
                 = FA (fa2 ⁀ tail fa))
          ∨ (∃ id: Identifier | arg = ID id
               • UpdateParamWithArg (comp, param, arg)
                 = FA (⟨id⟩ ⁀ tail fa))
          ∨ head fa ≠ var param
           ∧ UpdateParamWithArg (comp, param, arg) = comp))
```

```
UpdateMethodPropertiesArgsShare: ExprShareRelation × seq Expr × seq Variable
                         ⇸ ExprShareRelation
────────────────────────
∀ rel: ExprShareRelation; args: seq Expr; params: seq Variable
   • UpdateMethodPropertiesArgsShare (rel, args, params)
      = rel
      \ ∪ {  e1, e2: LExpr | (e1, e2) ∈ rel
           • {  n: 0 .. # args; e3, e4: LExpr
                | e3 = UpdateParamWithArg (e1, (params n), (args n))
                 ∧ e4
                    = UpdateParamWithArg (e2, (params n),
                                     (args n))
                 $\land (e1, e2) \neq (e3, e4) \bullet (e1, e2) \} \}$
      $\cup \bigcup \{ e1, e2: LExpr \mid (e1, e2) \in rel$
          $\bullet \{ n: 0 \mathrel{..} \# args; e3, e4: LExpr$
              $\mid e3 = UpdateParamWithArg (e1, (params\ n), (args\ n))$
              $\land e4$
                 $= UpdateParamWithArg (e2, (params\ n),$
                              $(args\ n)) \bullet (e3, e4) \} \}$
```

```
$UpdateMethodPropertiesArgsRef: MethodRefSet \times \operatorname{seq} Expr \times \operatorname{seq} Variable$
                 $\rightarrowtail MethodRefSet$
```

```
$\forall ref: MethodRefSet; args: \operatorname{seq} Expr; params: \operatorname{seq} Variable$
  $\bullet UpdateMethodPropertiesArgsRef (ref, args, params)$
    $= ref$
     $\setminus \bigcup \{ e1: LExpr; mrcs: \mathbb{P} MetaRefCon \mid (e1, mrcs) \in ref$
          $\bullet (\{ n: 0 \mathrel{..} \# args; e2: LExpr$
               $\mid e2$
                  $= UpdateParamWithArg (e1, (params\ n), (args\ n))$
                 $\land e1 \neq e2 \bullet (e1, mrcs) \}$
             $\cup \bigcup \{ n: 0 \mathrel{..} \# args$
                  $\bullet \{ mrc: mrcs; e3, e4: LExpr$
                      $\mid mrc = Erc\ e3$
                       $\land e4$
                          $= UpdateParamWithArg (e3,$
                                        $(params\ n),$
                                        $(args\ n))$
                       $\land e3 \neq e4 \bullet (e1, mrcs) \} \}) \}$
     $\cup \bigcup \{ e1: LExpr; mrcs: \mathbb{P} MetaRefCon \mid (e1, mrcs) \in ref$
          $\bullet \{ n: 0 \mathrel{..} \# args; e2: LExpr$
              $\mid e2 = UpdateParamWithArg (e1, (params\ n), (args\ n))$
              $\bullet (e2$
                $\mapsto mrcs$
                 $\setminus \{ mrc: mrcs; e3, e4: LExpr$
                      $\mid mrc = Erc\ e3$
                       $\land e4$
                          $= UpdateParamWithArg (e3,$
                                        $(params\ n),$
                                        $(args\ n))$
                       $\land e3 \neq e4 \bullet mrc \}$
                 $\cup \{ mrc: mrcs; e3, e4: LExpr$
                      $\mid mrc = Erc\ e3$
                       $\land e4$
                          $= UpdateParamWithArg (e3,$
                                        $(params\ n),$
                                        $(args\ n))$
                       $\land e3 \neq e4 \bullet Erc\ e4 \}) \} \}$
```

```
$UpdateMethodPropertiesArgs: Method \times \operatorname{seq} Expr \rightarrowtail MethodProperties$
```

```
$\forall m: Method; args: \operatorname{seq} Expr$
  $\bullet \exists rel: ExprShareRelation; ref: MethodRefSet \mid m.properties = (rel, ref)$
     $\bullet UpdateMethodPropertiesArgs (m, args)$
        $= UpdateMethodPropertiesArgsShare (rel, args, m.params)$
         $\mapsto UpdateMethodPropertiesArgsRef (ref, args, m.params)$
```

$$
\begin{array}{l}
\mathit{ApplyPossibleMethods}: \mathbb{P}\, \mathit{Method} \times \mathrm{seq}\, \mathit{Expr} \times \mathit{Env} \times \mathit{LExpr} \times \mathit{LExpr} \times \mathit{RefCon} \times \\
\qquad\qquad \mathit{SCJmSafeProgram} \\
\qquad\qquad \rightarrow\!\!\!\!\!+ \mathit{Env} \\
\hline
\forall \mathit{methods}: \mathbb{P}\, \mathit{Method}; \mathit{args}: \mathrm{seq}\, \mathit{Expr}; \mathit{env}: \mathit{Env}; \mathit{cexpr}, \mathit{lexpr}: \mathit{LExpr}; \\
\quad \mathit{rc}: \mathit{RefCon}; p: \mathit{SCJmSafeProgram} \\
\quad \bullet \exists \mathit{newLe}: \mathit{LExpr} \mid \mathit{newLe} = \mathit{MergeExprs}\, (\mathit{cexpr}, \mathit{lexpr}, p) \\
\qquad \bullet \mathit{methods} = \varnothing \\
\qquad\quad \land \mathit{ApplyPossibleMethods}\, (\mathit{methods}, \mathit{args}, \mathit{env}, \mathit{cexpr}, \mathit{lexpr}, \mathit{rc}, p) \\
\qquad\qquad = \mathit{env} \\
\qquad\quad \lor \mathit{methods} \neq \varnothing \\
\qquad\quad\ \land \mathit{ApplyPossibleMethods}\, (\mathit{methods}, \mathit{args}, \mathit{env}, \mathit{cexpr}, \mathit{lexpr}, \mathit{rc}, p) \\
\qquad\qquad = \mathit{DistEnvJoin}\, \{\ m: \mathit{methods} \\
\qquad\qquad\qquad\qquad \bullet (\mathit{ApplyMethodProperties}\, (m, \\
\qquad\qquad\qquad\qquad\qquad (\mathit{UpdateMethodPropertiesArgs}\, (m, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{args})), \\
\qquad\qquad\qquad\qquad\qquad \mathit{env}, \mathit{args}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{cexpr}, \mathit{newLe}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{rc}, p))\ \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{CalcEnvNewInstance}: \mathit{Env} \times \mathit{newInstance} \times \mathit{LExpr} \times \mathit{RefCon} \times \mathit{SCJmSafeProgram} \rightarrow\!\!\!\!\!+ \mathit{Env} \\
\hline
\forall \mathit{env}: \mathit{Env}; \mathit{nI}: \mathit{newInstance}; \mathit{cexpr}: \mathit{LExpr}; \mathit{rc}: \mathit{RefCon}; p: \mathit{SCJmSafeProgram} \\
\quad \bullet \exists \mathit{newLe}, e1: \mathit{LExpr}; \mathit{newMrc}: \mathit{MetaRefCon}; \mathit{constr}: \mathit{Method}; \\
\qquad \mathit{rel}: \mathit{ExprShareRelation}; \mathit{ref}: \mathit{ExprRefSet} \\
\qquad \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad\ \land \mathit{newLe} = \mathit{MergeExprs}\, (\mathit{cexpr}, \mathit{nI.le}, p) \\
\qquad\ \land (\mathit{nI.mrc} = \mathit{Erc}\ e1 \land \mathit{newMrc} = \mathit{Erc}\, (\mathit{MergeExprs}\, (\mathit{cexpr}, e1, p)) \\
\qquad\quad \lor \mathit{nI.mrc} \neq \mathit{Erc}\ e1 \land \mathit{newMrc} = \mathit{nI.mrc}) \\
\qquad\ \land \mathit{constr} = \mathit{GetConstr}\, (\mathit{nI.type.type}, \mathit{nI.args}, p) \\
\qquad \bullet \mathit{CalcEnvNewInstance}\, (\mathit{env}, \mathit{nI}, \mathit{cexpr}, \mathit{rc}, p) \\
\qquad\quad = \mathit{ApplyPossibleMethods}\, (\{\mathit{constr}\}, \mathit{nI.args}, \\
\qquad\qquad\qquad (\mathit{rel} \\
\qquad\qquad\qquad \mapsto \mathit{ExprRefUpdate}\, (\mathit{newLe}, \\
\qquad\qquad\qquad\qquad (\mathit{RCsFromMRC}\, (\mathit{newMrc}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{rc}, \mathit{ref}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{cexpr})), \\
\qquad\qquad\qquad\qquad \mathit{ref})), \mathit{cexpr}, \mathit{nI.le}, \\
\qquad\qquad\qquad \mathit{rc}, p)
\end{array}
$$

$$
\begin{array}{l}
\mathit{RemoveOutOfScopeVars}: \mathit{Env} \times \mathbb{P}\, \mathit{LExpr} \rightarrow\!\!\!\!\!+ \mathit{Env} \\
\hline
\forall \mathit{env}: \mathit{Env}; \mathit{vars}: \mathbb{P}\, \mathit{LExpr} \\
\quad \bullet \exists \mathit{rel}: \mathit{ExprShareRelation}; \mathit{ref}: \mathit{ExprRefSet} \mid \mathit{env} = (\mathit{rel}, \mathit{ref}) \\
\qquad \bullet \mathit{RemoveOutOfScopeVars}\, (\mathit{env}, \mathit{vars}) \\
\qquad\quad = \mathit{ExprShareRemoveSet}\, (\mathit{vars}, \mathit{rel}) \mapsto \mathit{ExprRefRemoveSet}\, (\mathit{vars}, \mathit{ref})
\end{array}
$$

$$
\begin{array}{l}
\mathit{CalcEnvMethod}: \mathit{Env} \times \mathit{methodCall} \times \mathit{LExpr} \times \mathit{RefCon} \times \mathit{SCJmSafeProgram} \rightarrow\!\!\!\!\!+ \mathit{Env} \\
\hline
\forall \mathit{env}: \mathit{Env}; \mathit{mc}: \mathit{methodCall}; \mathit{cexpr}: \mathit{LExpr}; \mathit{rc}: \mathit{RefCon}; p: \mathit{SCJmSafeProgram} \\
\quad \bullet \mathit{CalcEnvMethod}\, (\mathit{env}, \mathit{mc}, \mathit{cexpr}, \mathit{rc}, p) \\
\qquad = \mathit{RemoveOutOfScopeVars}\, ((\mathit{ApplyPossibleMethods}\, ((\mathit{GetMethodsFromSigs}\, (\mathit{mc.methods}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p)),
\end{array}
$$

```
                                    mc.args, env, cexpr,
                                    mc.le, rc, p)),
                        (∪ {   m: GetMethodsFromSigs (mc.methods, p)
                              • m.localVars  }))
```

```
CalcEnvGetMemArea: Env × getMemoryArea × LExpr × RefCon × SCJmSafeProgram
            ⇸ Env
─────────────────────────
∀ env: Env; gma: getMemoryArea; cexpr: LExpr; rc: RefCon; p: SCJmSafeProgram
  • ∃ rel: ExprShareRelation; ref: ExprRefSet ∣  env = (rel, ref)
    • ∃ newRef, newExpr: LExpr; erc: MetaRefCon
       ∣  newRef = MergeExprs (cexpr, gma.ref, p)
        ∧ (gma.e = This ∧ newExpr = cexpr
          ∨ gma.e ≠ This ∧ newExpr = MergeExprs (cexpr, gma.e, p))
        ∧ erc = Erc newExpr
      • CalcEnvGetMemArea (env, gma, cexpr, rc, p)
        = rel
          ↦ ExprRefUpdate (newRef,
                      (RCsFromMRC (erc, rc, ref, cexpr)),
                      ref)
```

```
CalcEnvCom: Env × Com × LExpr × RefCon × SCJmSafeProgram ⇸ Env
─────────────────────────
∀ env: Env; c: Com; cexpr: LExpr; rc: RefCon; p: SCJmSafeProgram
  • c = Skip ∧ CalcEnvCom (env, c, cexpr, rc, p) = env
    ∨ (∃ d: Dec
        • c = Decl d
          ∧ CalcEnvCom (env, c, cexpr, rc, p) = AddDecToEnv (env, d))
    ∨ (∃ nI: newInstance
        • c = NewInstance nI
          ∧ CalcEnvCom (env, c, cexpr, rc, p)
            = CalcEnvNewInstance (env, nI, cexpr, rc, p))
    ∨ (∃ c1: Com
        • c = Scope c1
          ∧ CalcEnvCom (env, c, cexpr, rc, p)
            = CalcEnvCom (env, c1, cexpr, rc, p))
    ∨ (∃ le: LExpr; re: Expr
        • c = Asgn (le, re)
          ∧ CalcEnvCom (env, c, cexpr, rc, p)
            = CalcEnvAssignment (env, le, re, cexpr, rc, p))
    ∨ (∃ c1, c2: Com
        • c = Seq (c1, c2)
          ∧ CalcEnvCom (env, c, cexpr, rc, p)
            = CalcEnvCom ((CalcEnvCom (env, c1, cexpr, rc, p)), c2,
                      cexpr, rc, p))
    ∨ (∃ e: Expr; c1, c2: Com
        • c = If (e, c1, c2)
          ∧ CalcEnvCom (env, c, cexpr, rc, p)
            = EnvJoin ((CalcEnvCom (env, c1, cexpr, rc, p)),
                    (CalcEnvCom (env, c2, cexpr, rc, p))))
    ∨ (∃ e: Expr; comSeq: seq Com
        • c = Switch (e, comSeq)
          ∧ CalcEnvCom (env, c, cexpr, rc, p)
            = DistEnvJoin {   c: ran comSeq
```

```
                              • (CalcEnvCom (env, c, cexpr, rc, p))  })
      ∨ (∃ c1, c2, c3: Com; exp: Expr
          • c = For (c1, exp, c2, c3)
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvCom ((CalcEnvCom (env, c1, cexpr, rc, p)),
                        (Seq (c2, c3)), cexpr, rc, p))
      ∨ (∃ mc: methodCall
          • c = MethodCall mc
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvMethod (env, mc, cexpr, rc, p))
      ∨ (∃ mc: methodCall
          • c = EnterPrivateMemory mc
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvMethod (env, mc, cexpr, (LowerRC rc), p))
      ∨ (∃ mrc: MetaRefCon; mc: methodCall; ref: ExprRefSet | ref = env.2
          • c = ExecuteInAreaOf (mrc, mc)
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = DistEnvJoin {  rc1: RCsFromMRC (mrc, rc, ref, cexpr)
                              • (CalcEnvMethod (env, mc, cexpr, rc1,
                                            p))  })
      ∨ (∃ mc: methodCall
          • c = ExecuteInOuterArea mc
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvMethod (env, mc, cexpr, (RaiseRC rc), p))
      ∨ (∃ gma: getMemoryArea
          • c = GetMemoryArea gma
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvGetMemArea (env, gma, cexpr, rc, p))
      ∨ (∃ c1, c2: Com; eseq: seq Expr; comseq: seq Com
          • c = Try (c1, eseq, comseq, c2)
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = EnvJoin ((EnvJoin ((CalcEnvCom (env, c1, cexpr, rc, p)),
                            (DistEnvJoin {  com: ran comseq
                                           • (CalcEnvCom (env,
                                                  com,
                                                  cexpr,
                                                  rc,
                                                  p))  }))),
                        (CalcEnvCom (env, c2, cexpr, rc, p))))
      ∨ (∃ e: Expr; c1: Com
          • c = While (e, c1)
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvCom (env, c1, cexpr, rc, p))
      ∨ (∃ e: Expr; c1: Com
          • c = DoWhile (c1, e)
            ∧ CalcEnvCom (env, c, cexpr, rc, p)
              = CalcEnvCom (env, c1, cexpr, rc, p))
```

```
  GetComSet: Com × SCJmSafeProgram ⇸ ℙ Com
  ─────────────────────────────────────
  ∀ c: Com; p: SCJmSafeProgram
    • c = Skip ∧ GetComSet (c, p) = {c}
      ∨ (∃ d: Dec • c = Decl d ∧ GetComSet (c, p) = {c})
      ∨ (∃ nI: newInstance • c = NewInstance nI ∧ GetComSet (c, p) = {c})
      ∨ (∃ c1: Com • c = Scope c1 ∧ GetComSet (c, p) = GetComSet (c1, p))
```

```
$\lor (\exists\, le: LExpr; re: Expr \bullet\ c = Asgn\ (le, re) \land GetComSet\ (c, p) = \{c\})$
$\lor (\exists\, c1, c2: Com$
    $\bullet\ c = Seq\ (c1, c2)$
      $\land\ GetComSet\ (c, p) = GetComSet\ (c1, p) \cup GetComSet\ (c2, p))$
$\lor (\exists\, e: Expr; c1, c2: Com$
    $\bullet\ c = If\ (e, c1, c2)$
      $\land\ GetComSet\ (c, p) = GetComSet\ (c1, p) \cup GetComSet\ (c2, p))$
$\lor (\exists\, e: Expr; comSeq: \mathrm{seq}\ Com$
    $\bullet\ c = Switch\ (e, comSeq)$
      $\land\ GetComSet\ (c, p)$
        $= \bigcup \{\ c1: \mathrm{ran}\ comSeq \bullet\ (GetComSet\ (c1, p))\ \})$
$\lor (\exists\, c1, c2, c3: Com; exp: Expr$
    $\bullet\ c = For\ (c1, exp, c2, c3)$
      $\land\ GetComSet\ (c, p)$
        $= GetComSet\ (c1, p) \cup GetComSet\ (c2, p)$
          $\cup\ GetComSet\ (c3, p))$
$\lor (\exists\, mc: methodCall$
    $\bullet\ c = MethodCall\ mc$
      $\land\ GetComSet\ (c, p)$
        $= \bigcup \{\ m: GetMethodsFromSigs\ (mc.methods, p)$
            $\bullet\ (GetComSet\ (m.body, p))\ \})$
$\lor (\exists\, mc: methodCall$
    $\bullet\ c = EnterPrivateMemory\ mc$
      $\land\ GetComSet\ (c, p)$
        $= \bigcup \{\ m: GetMethodsFromSigs\ (mc.methods, p)$
            $\bullet\ (GetComSet\ (m.body, p))\ \})$
$\lor (\exists\, mrc: MetaRefCon; mc: methodCall$
    $\bullet\ c = ExecuteInAreaOf\ (mrc, mc)$
      $\land\ GetComSet\ (c, p)$
        $= \bigcup \{\ m: GetMethodsFromSigs\ (mc.methods, p)$
            $\bullet\ (GetComSet\ (m.body, p))\ \})$
$\lor (\exists\, mc: methodCall$
    $\bullet\ c = ExecuteInOuterArea\ mc$
      $\land\ GetComSet\ (c, p)$
        $= \bigcup \{\ m: GetMethodsFromSigs\ (mc.methods, p)$
            $\bullet\ (GetComSet\ (m.body, p))\ \})$
$\lor (\exists\, gma: getMemoryArea$
    $\bullet\ c = GetMemoryArea\ gma \land GetComSet\ (c, p) = \{c\})$
$\lor (\exists\, c1, c2: Com; eseq: \mathrm{seq}\ Expr; comseq: \mathrm{seq}\ Com$
    $\bullet\ c = Try\ (c1, eseq, comseq, c2)$
      $\land\ GetComSet\ (c, p)$
        $= GetComSet\ (c1, p) \cup GetComSet\ (c2, p)$
          $\cup \bigcup \{\ com: \mathrm{ran}\ comseq \bullet\ (GetComSet\ (com, p))\ \})$
$\lor (\exists\, e: Expr; c1: Com$
    $\bullet\ c = While\ (e, c1) \land GetComSet\ (c, p) = GetComSet\ (c1, p))$
$\lor (\exists\, c1: Com; e: Expr$
    $\bullet\ c = DoWhile\ (c1, e) \land GetComSet\ (c, p) = GetComSet\ (c1, p))$
```

```
$GetHandlerExpr: SCJmSafeProgram \times Handler \times Mission \pfun LExpr$
--------------------------------
$\forall\, p: SCJmSafeProgram; h: Handler; m: Mission$
  $\bullet\ \mathbf{let}\ coms == GetComSet\ (m.initialize, p)$
    $\bullet\ \exists\, nI: newInstance; mc: methodCall$
        $\mid\ nI.type.type = h.name$
          $\land\ mc.le = nI.le$
          ∧ mc.name = register
          ∧ NewInstance nI ∈ coms
          ∧ MethodCall mc ∈ coms • GetHandlerExpr (p, h, m) = nI.le
```

```
GetHandlerExprs: SCJmSafeProgram × Mission ⇸ ℙ LExpr
─────────────────────────────
∀ p: SCJmSafeProgram; m: Mission
  • let coms == GetComSet (m.initialize, p);
      handlerNames == { h: p.handlers • h.name }
    • GetHandlerExprs (p, m)
       = { nI: newInstance; mc: methodCall
            | nI.type.type ∈ handlerNames
             ∧ mc.le = nI.le
             ∧ mc.name = register
             ∧ NewInstance nI ∈ coms
             ∧ MethodCall mc ∈ coms • nI.le }
```

```
GetMissionExpr: SCJmSafeProgram × Mission ⇸ LExpr
─────────────────────────────
∀ p: SCJmSafeProgram; m: Mission
  • let coms == GetComSet (p.missionSeq.getNextMission, p)
    • ∃ nI: newInstance; e1, e2: Expr; c: Com; v: Variable
        | nI.type.type = m.name
         ∧ c = Asgn (e1, e2)
         ∧ e1 = ID (var v)
         ∧ v.name = Result
         ∧ e2 = nI.le
         ∧ NewInstance nI ∈ coms
         ∧ c ∈ coms • GetMissionExpr (p, m) = nI.le
```

```
GetMissionExprs: SCJmSafeProgram ⇸ ℙ LExpr
─────────────────────────────
∀ p: SCJmSafeProgram
  • let coms == GetComSet (p.missionSeq.getNextMission, p);
      missionNames == { m: p.missions • m.name }
    • GetMissionExprs p
       = { nI: newInstance; e1, e2: Expr; c: Com; v: Variable
            | nI.type.type ∈ missionNames
             ∧ c = Asgn (e1, e2)
             ∧ e1 = ID (var v)
             ∧ v.name = Result
             ∧ e2 = nI.le
             ∧ NewInstance nI ∈ coms
             ∧ c ∈ coms • nI.le }
```

```
GetMissionSeqExpr: SCJmSafeProgram × MissionSeq ⇸ LExpr
─────────────────────────────
∀ p: SCJmSafeProgram; ms: MissionSeq
  • let coms == GetComSet (p.safelet.getSequencer, p)
    • ∃ nI: newInstance; e1, e2: Expr; c: Com; v: Variable
        | nI.type.type = ms.name
         ∧ c = Asgn (e1, e2)
```

$\land e1 = ID\ (var\ v)$
$\land v.name = Result$
$\land e2 = nI.le$
$\land NewInstance\ nI \in coms$
$\land c \in coms \bullet\ GetMissionSeqExpr\ (p, ms) = nI.le$

$LocalVars: Com \rightarrowtail \mathbb{P}\ LExpr$

$\forall c: Com$
$\bullet\ c = Skip \land LocalVars\ c = \varnothing$
$\lor (\exists d: Dec \bullet\ c = Decl\ d \land LocalVars\ c = \{ID\ (var\ d.var)\})$
$\lor (\exists nI: newInstance \bullet\ c = NewInstance\ nI \land LocalVars\ c = \varnothing)$
$\lor (\exists c1: Com \bullet\ c = Scope\ c1 \land LocalVars\ c = LocalVars\ c1)$
$\lor (\exists le: LExpr; re: Expr \bullet\ c = Asgn\ (le, re) \land LocalVars\ c = \varnothing)$
$\lor (\exists c1, c2: Com$
$\bullet\ c = Seq\ (c1, c2) \land LocalVars\ c = LocalVars\ c1 \cup LocalVars\ c2)$
$\lor (\exists e: Expr; c1, c2: Com$
$\bullet\ c = If\ (e, c1, c2) \land LocalVars\ c = LocalVars\ c1 \cup LocalVars\ c2)$
$\lor (\exists e: Expr; comSeq: \mathrm{seq}\ Com$
$\bullet\ c = Switch\ (e, comSeq)$
$\land LocalVars\ c = \bigcup \{\ c1: \mathrm{ran}\ comSeq \bullet\ (LocalVars\ c1)\ \})$
$\lor (\exists c1, c2, c3: Com; exp: Expr$
$\bullet\ c = For\ (c1, exp, c2, c3)$
$\land LocalVars\ c = LocalVars\ c1 \cup LocalVars\ c2 \cup LocalVars\ c3)$
$\lor (\exists mc: methodCall \bullet\ c = MethodCall\ mc \land LocalVars\ c = \varnothing)$
$\lor (\exists mc: methodCall \bullet\ c = EnterPrivateMemory\ mc \land LocalVars\ c = \varnothing)$
$\lor (\exists mrc: MetaRefCon; mc: methodCall$
$\bullet\ c = ExecuteInAreaOf\ (mrc, mc) \land LocalVars\ c = \varnothing)$
$\lor (\exists mc: methodCall \bullet\ c = ExecuteInOuterArea\ mc \land LocalVars\ c = \varnothing)$
$\lor (\exists gma: getMemoryArea \bullet\ c = GetMemoryArea\ gma \land LocalVars\ c = \varnothing)$
$\lor (\exists c1, c2: Com; eseq: \mathrm{seq}\ Expr; comseq: \mathrm{seq}\ Com$
$\bullet\ c = Try\ (c1, eseq, comseq, c2)$
$\land LocalVars\ c$
$= LocalVars\ c1 \cup LocalVars\ c2$
$\cup \bigcup \{\ com: \mathrm{ran}\ comseq \bullet\ (LocalVars\ com)\ \})$
$\lor (\exists e: Expr; c1: Com \bullet\ c = While\ (e, c1) \land LocalVars\ c = LocalVars\ c1)$
$\lor (\exists c1: Com; e: Expr$
$\bullet\ c = DoWhile\ (c1, e) \land LocalVars\ c = LocalVars\ c1)$

$CalcEnvHandler: Env \times Handler \times LExpr \times SCJmSafeProgram \rightarrowtail Env$

$\forall env: Env; h: Handler; cexpr: LExpr; p: SCJmSafeProgram$
$\bullet\ CalcEnvHandler\ (env, h, cexpr, p)$
$= RemoveExprSetEnv\ ((LocalVars\ h.hAe),$
$(CalcEnvCom\ (env, h.hAe, cexpr, (PRMem\ h.name),$
$p)))$

$CalcEnvHandlers: Env \times Mission \times \mathbb{P}\ Handler \times SCJmSafeProgram \rightarrowtail Env$

$\forall env: Env; m: Mission; handlers: \mathbb{P}\ Handler; p: SCJmSafeProgram$
$\bullet\ CalcEnvHandlers\ (env, m, handlers, p)$
$= DistEnvJoin\ \{\ h: handlers$
$\bullet\ (CalcEnvHandler\ (env, h,$

```
                                    (GetHandlerExpr (p, h, m)),
                                    p)) }
```

```
GetHandlers: SCJmSafeProgram × ℙ Name ⇸ ℙ Handler
─────────────────────
∀ p: SCJmSafeProgram; names: ℙ Name
  • GetHandlers (p, names) = { h: p.handlers | h.name ∈ names • h }
```

```
CalcEnvMission: Env × Mission × LExpr × SCJmSafeProgram ⇸ Env
─────────────────────
∀ env: Env; m: Mission; cexpr: LExpr; p: SCJmSafeProgram
  • let initializeEnv == CalcEnvCom (env, m.initialize, cexpr, MMem, p)
    • let handlersEnv ==
            CalcEnvHandlers ((RemoveExprSetEnv ((LocalVars m.initialize
                                   \ GetHandlerExprs (p,
                                                  m)),
                                   initializeEnv)), m,
                     (GetHandlers (p, m.handlers)), p)
      • let cleanUpEnv ==
              CalcEnvCom ((RemoveExprSetEnv ((LocalVars m.initialize),
                                   handlersEnv)), m.cleanUp,
                     cexpr, MMem, p)
        • CalcEnvMission (env, m, cexpr, p)
          = RemoveExprSetEnv ((LocalVars m.cleanUp), cleanUpEnv)
```

```
CalcEnvMissions: Env × ℙ Mission × SCJmSafeProgram ⇸ Env
─────────────────────
∀ env: Env; missions: ℙ Mission; p: SCJmSafeProgram
  • CalcEnvMissions (env, missions, p)
    = DistEnvJoin { m: missions
                    • (CalcEnvMission (env, m, (GetMissionExpr (p, m)),
                                    p)) }
```

```
CalcEnvMissionSeq: Env × MissionSeq × SCJmSafeProgram ⇸ Env
─────────────────────
∀ env: Env; ms: MissionSeq; p: SCJmSafeProgram
  • let getNextMissionEnv ==
          CalcEnvCom (env, ms.getNextMission, (GetMissionSeqExpr (p, ms)),
                MMem, p)
    • CalcEnvMissionSeq (env, ms, p)
      = RemoveExprSetEnv ((LocalVars ms.getNextMission),
                 (CalcEnvMissions (getNextMissionEnv,
                                p.missions, p)))
```

```
ExprShareAddProperties: LExpr × LExpr × MethodProperties ⇸ MethodProperties
─────────────────────
∀ le₁, le₂: LExpr; properties: MethodProperties
  • ∃ rel: ExprShareRelation; ref: MethodRefSet | properties = (rel, ref)
    • ExprShareAddProperties (le₁, le₂, properties)
      = ExprShareAdd (le₁, le₂, rel) ↦ ref
```

$$\begin{array}{|l}
\mathit{ExprShareAddSetProperties}\colon \mathbb{P}\,(\mathit{LExpr} \times \mathit{LExpr}) \times \mathit{MethodProperties}\\
\qquad\qquad\qquad \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodProperties}\\
\hline
\forall\, \mathit{set}\colon \mathbb{P}\,(\mathit{LExpr} \times \mathit{LExpr});\ \mathit{properties}\colon \mathit{MethodProperties}\\
\quad \bullet\ \exists\, \mathit{rel}\colon \mathit{ExprShareRelation};\ \mathit{ref}\colon \mathit{MethodRefSet} \mid \mathit{properties} = (\mathit{rel}, \mathit{ref})\\
\qquad \bullet\ \mathit{ExprShareAddSetProperties}\,(\mathit{set}, \mathit{properties})\\
\qquad\quad = (\mathit{rel}\\
\qquad\qquad \cup \bigcup \{\ \mathit{le}_1, \mathit{le}_2\colon \mathit{LExpr} \mid (\mathit{le}_1, \mathit{le}_2) \in \mathit{set}\\
\qquad\qquad\qquad \bullet\ \{(\mathit{le}_1 \mapsto \mathit{le}_2), (\mathit{le}_2 \mapsto \mathit{le}_1)\}\ \})^*\\
\qquad\quad \mapsto \mathit{ref}
\end{array}$$

$$\begin{array}{|l}
\mathit{MethodRefAdd}\colon \mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon} \times \mathit{MethodRefSet} \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodRefSet}\\
\hline
\forall\, \mathit{le}\colon \mathit{LExpr};\ \mathit{mrcs}\colon \mathbb{P}\,\mathit{MetaRefCon};\ \mathit{ref}\colon \mathit{MethodRefSet}\\
\quad \bullet\ \mathit{MethodRefAdd}\,(\mathit{le}, \mathit{mrcs}, \mathit{ref}) = \mathit{ref} \oplus \{(\mathit{le} \mapsto \mathit{mrcs})\}
\end{array}$$

$$\begin{array}{|l}
\mathit{MethodRefUpdate}\colon \mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon} \times \mathit{MethodRefSet} \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodRefSet}\\
\hline
\forall\, \mathit{le}\colon \mathit{LExpr};\ \mathit{mrcs}\colon \mathbb{P}\,\mathit{MetaRefCon};\ \mathit{ref}\colon \mathit{MethodRefSet}\\
\quad \bullet\ \mathit{MethodRefUpdate}\,(\mathit{le}, \mathit{mrcs}, \mathit{ref}) = \mathit{ref} \oplus \{(\mathit{le} \mapsto \mathit{ref}\ \mathit{le} \cup \mathit{mrcs})\}
\end{array}$$

$$\begin{array}{|l}
\mathit{MethodRefUpdateSet}\colon \mathbb{P}\,(\mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon}) \times \mathit{MethodRefSet} \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodRefSet}\\
\hline
\forall\, \mathit{set}\colon \mathbb{P}\,(\mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon});\ \mathit{ref}\colon \mathit{MethodRefSet}\\
\quad \bullet\ \mathit{MethodRefUpdateSet}\,(\mathit{set}, \mathit{ref})\\
\qquad = \mathit{ref}\\
\qquad\quad \oplus \{\ \mathit{le}\colon \mathit{LExpr};\ \mathit{mrcs}\colon \mathbb{P}\,\mathit{MetaRefCon} \mid (\mathit{le}, \mathit{mrcs}) \in \mathit{set}\\
\qquad\qquad \bullet\ (\mathit{le} \mapsto \mathit{ref}\ \mathit{le} \cup \mathit{mrcs})\ \}
\end{array}$$

$$\begin{array}{|l}
\mathit{MethodRefAddProperties}\colon \mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon} \times \mathit{MethodProperties}\\
\qquad\qquad\qquad \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodProperties}\\
\hline
\forall\, \mathit{le}\colon \mathit{LExpr};\ \mathit{mrcs}\colon \mathbb{P}\,\mathit{MetaRefCon};\ \mathit{properties}\colon \mathit{MethodProperties}\\
\quad \bullet\ \exists\, \mathit{rel}\colon \mathit{ExprShareRelation};\ \mathit{ref}\colon \mathit{MethodRefSet} \mid \mathit{properties} = (\mathit{rel}, \mathit{ref})\\
\qquad \bullet\ \mathit{MethodRefAddProperties}\,(\mathit{le}, \mathit{mrcs}, \mathit{properties})\\
\qquad\quad = \mathit{rel} \mapsto \mathit{MethodRefAdd}\,(\mathit{le}, \mathit{mrcs}, \mathit{ref})
\end{array}$$

$$\begin{array}{|l}
\mathit{MethodRefUpdateProperties}\colon \mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon} \times \mathit{MethodProperties}\\
\qquad\qquad\qquad \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodProperties}\\
\hline
\forall\, \mathit{le}\colon \mathit{LExpr};\ \mathit{mrcs}\colon \mathbb{P}\,\mathit{MetaRefCon};\ \mathit{properties}\colon \mathit{MethodProperties}\\
\quad \bullet\ \exists\, \mathit{rel}\colon \mathit{ExprShareRelation};\ \mathit{ref}\colon \mathit{MethodRefSet} \mid \mathit{properties} = (\mathit{rel}, \mathit{ref})\\
\qquad \bullet\ \mathit{MethodRefUpdateProperties}\,(\mathit{le}, \mathit{mrcs}, \mathit{properties})\\
\qquad\quad = \mathit{rel} \mapsto \mathit{MethodRefUpdate}\,(\mathit{le}, \mathit{mrcs}, \mathit{ref})
\end{array}$$

$$\begin{array}{|l}
\mathit{MethodRefUpdateSetProperties}\colon \mathbb{P}\,(\mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon}) \times \mathit{MethodProperties}\\
\qquad\qquad\qquad \mathrel{\mkern6mu\mapstochar\mkern-6mu\rightarrow} \mathit{MethodProperties}\\
\hline
\forall\, \mathit{set}\colon \mathbb{P}\,(\mathit{LExpr} \times \mathbb{P}\,\mathit{MetaRefCon});\ \mathit{properties}\colon \mathit{MethodProperties}\\
\quad \bullet\ \exists\, \mathit{rel}\colon \mathit{ExprShareRelation};\ \mathit{ref}\colon \mathit{MethodRefSet} \mid \mathit{properties} = (\mathit{rel}, \mathit{ref})
\end{array}$$

```
        • $MethodRefUpdateSetProperties\ (set, properties)$
          $= rel$
           $\mapsto ref$
             $\oplus \{\ le: LExpr; mrcs: \mathbb{P}\ MetaRefCon \mid (le, mrcs) \in set$
                 $\bullet\ (le \mapsto ref\ le \cup mrcs)\ \}$
```

```
$MethodRefJoin: MethodRefSet \times MethodRefSet \rightarrow MethodRefSet$
────────────────
$\forall ref_1, ref_2: MethodRefSet$
  • $MethodRefJoin\ (ref_1, ref_2)$
     $= (\mathrm{dom}\ ref_1 \setminus \mathrm{dom}\ ref_2) \lhd ref_1 \cup (\mathrm{dom}\ ref_2 \setminus \mathrm{dom}\ ref_1) \lhd ref_2$
    $\cup \{\ le: \mathrm{dom}\ ref_1 \cap \mathrm{dom}\ ref_2 \bullet (le \mapsto ref_1\ le \cup ref_2\ le)\ \}$
```

```
$DistMethodRefJoin: \mathbb{F}\ MethodRefSet \rightarrow MethodRefSet$
────────────────
$\forall mrefsets: \mathbb{F}\ MethodRefSet$
  • $\exists ref: mrefsets$
      • $mrefsets \neq \emptyset$
       $\wedge DistMethodRefJoin\ mrefsets$
          $= MethodRefJoin\ (ref, (DistMethodRefJoin\ (mrefsets \setminus \{ref\})))$
       $\vee mrefsets = \emptyset \wedge DistMethodRefJoin\ mrefsets = \emptyset$
```

```
$MethodPropertiesJoin: MethodProperties \times MethodProperties \rightarrow MethodProperties$
────────────────
$\forall p_1, p_2: MethodProperties$
  • $\exists rel_1, rel_2: ExprShareRelation; ref_1, ref_2: MethodRefSet$
     $\mid (rel_1, ref_1) = p_1 \wedge (rel_2, ref_2) = p_2$
      • $MethodPropertiesJoin\ (p_1, p_2)$
         $= rel_1 \cup rel_2 \mapsto MethodRefJoin\ (ref_1, ref_2)$
```

```
$DistMethodPropertiesJoin: \mathbb{F}\ MethodProperties \rightarrow MethodProperties$
────────────────
$\forall properties: \mathbb{F}\ MethodProperties$
  • $properties = \emptyset$
   $\wedge (\exists p: properties$
       • $DistMethodPropertiesJoin\ properties$
          $= MethodPropertiesJoin\ (p,$
                              $(DistMethodPropertiesJoin\ (properties$
                                           $\setminus \{p\}))))$
   $\vee properties = \emptyset \wedge DistMethodPropertiesJoin\ properties = (\emptyset, \emptyset)$
```

```
$RaiseMRC: MetaRefCon \rightarrow MetaRefCon$
────────────────
$\forall mrc: MetaRefCon$
  • $\exists n: \mathbb{N}$
      • $mrc = CurrentPlus\ n \wedge RaiseMRC\ mrc = CurrentPlus\ (n + 1)$
       $\vee mrc = Current \wedge RaiseMRC\ mrc = CurrentPlus\ 0$
       $\vee mrc = CurrentPrivate\ n$
         $\wedge n > 0$
         $\wedge RaiseMRC\ mrc = CurrentPrivate\ (n - 1)$
       $\vee mrc = CurrentPrivate\ n \wedge n = 0 \wedge RaiseMRC\ mrc = Current$
```

```
RaiseMRCBy: MetaRefCon × ℕ ⇸ MetaRefCon
────────────────────
∀ mrc: MetaRefCon; count: ℕ
  • count > 0
    ∧ RaiseMRCBy (mrc, count) = RaiseMRCBy ((RaiseMRC mrc), (count - 1))
    ∨ count = 0 ∧ RaiseMRCBy (mrc, count) = mrc
```

```
LowerMRC: MetaRefCon ⇸ MetaRefCon
────────────────────
∀ mrc: MetaRefCon
  • ∃ n: ℕ
      • mrc = CurrentPlus n ∧ n > 0 ∧ LowerMRC mrc = CurrentPlus (n - 1)
        ∨ mrc = CurrentPlus n ∧ n = 0 ∧ LowerMRC mrc = Current
        ∨ mrc = Current ∧ LowerMRC mrc = CurrentPrivate 0
        ∨ mrc = CurrentPrivate n ∧ LowerMRC mrc = CurrentPrivate (n + 1)
```

```
LowerMRCBy: MetaRefCon × ℕ ⇸ MetaRefCon
────────────────────
∀ mrc: MetaRefCon; count: ℕ
  • count > 0
    ∧ LowerMRCBy (mrc, count) = LowerMRCBy ((LowerMRC mrc), (count - 1))
    ∨ count = 0 ∧ LowerMRCBy (mrc, count) = mrc
```

```
AnalyseMetaRefCon: MetaRefCon × MetaRefCon × MethodRefSet ⇸ ℙ MetaRefCon
────────────────────
∀ nImrc, mrc: MetaRefCon; ref: MethodRefSet
  • (∃ n: ℕ | nImrc = CurrentPlus n
      • (∃ n1: ℕ | mrc = CurrentPlus n1
          • AnalyseMetaRefCon (nImrc, mrc, ref)
            = {CurrentPlus (n + n1)})
        ∨ mrc = Current
         ∧ AnalyseMetaRefCon (nImrc, mrc, ref) = {CurrentPrivate n}
        ∨ (∃ n1: ℕ | mrc = CurrentPrivate n1
            • (n - n1 > 0
              ∧ AnalyseMetaRefCon (nImrc, mrc, ref)
                 = {CurrentPlus (n - n1)}
              ∨ n - n1 < 0
               ∧ AnalyseMetaRefCon (nImrc, mrc, ref)
                  = {CurrentPrivate (n1 - n)}
              ∨ n - n1 = 0
               ∧ AnalyseMetaRefCon (nImrc, mrc, ref) = {Current})))
    ∨ nImrc = Current ∧ AnalyseMetaRefCon (nImrc, mrc, ref) = {mrc}
    ∨ (∃ n: ℕ | nImrc = CurrentPrivate n
        • ((∃ n1: ℕ | mrc = CurrentPlus n1
              • n1 - n > 0
              ∧ AnalyseMetaRefCon (nImrc, mrc, ref)
                 = {CurrentPlus (n1 - n)}
              ∨ n1 - n < 0
               ∧ AnalyseMetaRefCon (nImrc, mrc, ref)
                  = {CurrentPrivate (n1 - n)}
              ∨ n1 - n = 0
               ∧ AnalyseMetaRefCon (nImrc, mrc, ref) = {Current})
          ∨ mrc = Current
│              ∧ AnalyseMetaRefCon (nImrc, mrc, ref) = {CurrentPrivate n}
│            ∨ (∃ n1: ℕ | mrc = CurrentPrivate n1
│                  • AnalyseMetaRefCon (nImrc, mrc, ref)
│                     = {CurrentPrivate (n1 + n)})))
│      ∨ (∃ e: LExpr | nImrc = Erc e
│            • AnalyseMetaRefCon (nImrc, mrc, ref) = ref e)
```

```
│ GetDecMetaRefCon: Dec ⇸ ℙ MetaRefCon
├──────────────────────
│ ∀ d: Dec
│   • d.var.varType.isPrimitive = True ∧ GetDecMetaRefCon d = {Rcs {Prim}}
│     ∨ d.var.varType.isPrimitive = False ∧ GetDecMetaRefCon d = {}
```

```
│ AddDecToMethodProperties: MethodProperties × Dec ⇸ MethodProperties
├──────────────────────
│ ∀ properties: MethodProperties; d: Dec
│   • AddDecToMethodProperties (properties, d)
│      = MethodRefAddProperties ((ID (var d.var)), (GetDecMetaRefCon d),
│                    (ExprShareAddProperties ((ID (var d.var)),
│                                  (ID (var d.var)),
│                                  properties)))
```

```
│ AddDecsToMethodProperties: MethodProperties × ℙ Dec ⇸ MethodProperties
├──────────────────────
│ ∀ properties: MethodProperties; decs: ℙ Dec
│   • (∃ d: decs
│        • AddDecsToMethodProperties (properties, decs)
│          = AddDecsToMethodProperties ((AddDecToMethodProperties (properties,
│                                                d)),
│                       (decs \ {d})))
│     ∨ decs = ∅ ∧ AddDecsToMethodProperties (properties, decs) = properties
```

```
│ AddAsgnFieldsProperties: LExpr × Expr × MethodProperties ⇸ MethodProperties
├──────────────────────
│ ∀ properties: MethodProperties; le: LExpr; re: Expr
│   • ∃ rel: ExprShareRelation; ref: MethodRefSet | properties = (rel, ref)
│      • AddAsgnFieldsProperties (le, re, properties)
│        = ExprShareAddSet ({ e1, e2: LExpr
│                       | (e1, e2) ∈ rel
│                         ∧ PrefixOf (re, e1) = True
│                       • ((MergeShareExprExprs (le, re, e1)),
│                         e1) }, rel)
│           ↦ MethodRefUpdateSet ({ e1, e2: LExpr
│                          | (e1, e2) ∈ rel
│                            ∧ PrefixOf (re, e1) = True
│                          • ((MergeShareExprExprs (le, re,
│                                          e1)),
│                            (ref e1)) }, ref)
```

```
│ AddAsgnFieldsPropertiesSet: ℙ (LExpr × Expr) × MethodProperties
│                  ⇸ MethodProperties
```

```
∀ shareSet: ℙ (LExpr × LExpr); properties: MethodProperties
  • ∃ rel: ExprShareRelation; ref: MethodRefSet | properties = (rel, ref)
      • AddAsgnFieldsPropertiesSet (shareSet, properties)
        = ∪ {  le: LExpr; re: Expr |  (le, re) ∈ shareSet
              • (ExprShareAddSet ({  e1, e2: LExpr
                                  | (e1, e2) ∈ rel
                                    ∧ PrefixOf (re, e1) = True
                                  • ((MergeShareExprExprs (le,
                                                          re,
                                                          e1)),
                                  e1)  }, rel))  }
          ↦ DistMethodRefJoin {  le: LExpr; re: Expr
                              | (le, re) ∈ shareSet
                              • (MethodRefUpdateSet ({  e1,
                                                      e2: LExpr
                                                     | (e1,
                                                       e2)
                                                       ∈ rel
                                                      ∧ PrefixOf (re,
                                                                  e1)
                                                        = True
                                                     • ((MergeShareExprExprs (le,
                                                                             re,
                                                                             e1)),
                                                     (ref e1))  },
                                                  ref))  }
```

```
AddAsgnFieldsMethodRefSet: ℙ (LExpr × Expr) × ExprShareRelation × MethodRefSet
                      ⇸ MethodRefSet
─────────────────────
∀ shareSet: ℙ (LExpr × LExpr); rel: ExprShareRelation; ref: MethodRefSet
  • AddAsgnFieldsMethodRefSet (shareSet, rel, ref)
    = DistMethodRefJoin {  le: LExpr; re: Expr |  (le, re) ∈ shareSet
                        • (MethodRefUpdateSet ({  e1, e2: LExpr
                                               | (e1, e2) ∈ rel
                                                ∧ PrefixOf (re,
                                                            e1)
                                                  = True
                                               • ((MergeShareExprExprs (le,
                                                                       re,
                                                                       e1)),
                                               (ref e1))  },
                                            ref))  }
```

```
UpdateEqualExprsProperties: LExpr × Expr × MethodProperties ⇸ MethodProperties
─────────────────────
∀ properties: MethodProperties; le: LExpr; re: Expr
  • ∃ rel: ExprShareRelation; ref: MethodRefSet | properties = (rel, ref)
      • (∃ fa: FieldAccess |  le = FA fa
          • (let shareSet  ==
                {  e1, e2, equale: LExpr
                  | (e1, e2) ∈ rel
                    ∧ FA (front fa) = e1
                  $\wedge$ $equale = MergeShareExprExprs\ (e2, e1, le)$
                $\bullet$ $(equale, re)$ };
          $refSet$ ==
            { $e1, e2, equale: LExpr$
              | $(e1, e2) \in rel$
                $\wedge$ $FA\ (front\ fa) = e1$
                $\wedge$ $equale = MergeShareExprExprs\ (e2, e1, le)$
              $\bullet$ $(equale, ref\ re)$ }
        $\bullet$ $UpdateEqualExprsProperties\ (le, re, properties)$
            $= AddAsgnFieldsPropertiesSet\ (shareSet,$
                              $(ExprShareAddSet\ (shareSet,$
                                        $rel)$
                              $\mapsto MethodRefUpdateSet\ (refSet,$
                                        $ref)))))$
    $\vee$ ($\exists$ $ae: ArrayElement$ | $le = ID\ (arrayElement\ ae)$
      $\bullet$ (let $shareSet$ ==
                { $e1, e2, equale: LExpr; v: Variable;$
                  $fa: FieldAccess$
                  | $(e1, e2) \in rel$
                    $\wedge$ $v.name = ae.name$
                    $\wedge$ $last\ fa = var\ v$
                    $\wedge$ $(e1 = ID\ (var\ v) \vee e1 = FA\ fa)$
                    $\wedge$ $equale$
                       $= MergeShareExprExprs\ (e2, e1, le)$
                  $\bullet$ $(equale, re)$ };
          $refSet$ ==
                { $e1, e2, equale: LExpr; v: Variable;$
                  $fa: FieldAccess$
                  | $(e1, e2) \in rel$
                    $\wedge$ $v.name = ae.name$
                    $\wedge$ $last\ fa = var\ v$
                    $\wedge$ $(e1 = ID\ (var\ v) \vee e1 = FA\ fa)$
                    $\wedge$ $equale$
                       $= MergeShareExprExprs\ (e2, e1, le)$
                  $\bullet$ $(equale, ref\ re)$ }
        $\bullet$ $UpdateEqualExprsProperties\ (le, re, properties)$
            $= AddAsgnFieldsPropertiesSet\ (shareSet,$
                              $(ExprShareAddSet\ (shareSet,$
                                        $rel)$
                              $\mapsto MethodRefUpdateSet\ (refSet,$
                                        $ref)))))$
```

```
$UpdateEqualExprsPropertiesSet: \mathbb{P}\ (LExpr \times Expr) \times MethodProperties$
                    $\rightarrow\!\!\!\!\!+ MethodProperties$
-----------------------------
$\forall$ $properties: MethodProperties; set: \mathbb{P}\ (LExpr \times Expr)$
  $\bullet$ $\exists$ $rel: ExprShareRelation; ref: MethodRefSet$ | $properties = (rel, ref)$
    $\bullet$ $UpdateEqualExprsPropertiesSet\ (set, properties)$
      $= \bigcup$ { $le: LExpr; re: Expr$ | $(le, re) \in set$
        $\bullet$ ($\bigcup$ { $fa: FieldAccess$ | $le = FA\ fa$
            $\bullet$ (let $shareSet$ ==
                    { $e1, e2, equale: LExpr$
                      | $(e1, e2) \in rel$
                        $\wedge$ $FA\ (front\ fa) = e1$
                        $\wedge$ $equale$
                        = MergeShareExprExprs (e2,
                                               e1,
                                               le)
                    • (equale, re) }
              • (AddAsgnFieldsShareSet (shareSet,
                                        (ExprShareAddSet (shareSet,
                                                          rel))))) }
       ∪ ∪ {  ae: ArrayElement | le = ID (arrayElement ae)
            • (let shareSet ==
                   {  e1, e2, equale: LExpr;
                      v: Variable; fa: FieldAccess
                      | (e1, e2) ∈ rel
                        ∧ v.name = ae.name
                        ∧ last fa = var v
                        ∧ (e1 = ID (var v)
                           ∨ e1 = FA fa)
                        ∧ equale
                          = MergeShareExprExprs (e2,
                                                 e1,
                                                 le)
                      • (equale, re) }
                • (AddAsgnFieldsShareSet (shareSet,
                                          (ExprShareAddSet (shareSet,
                                                            rel))))) }) }
   ↦ DistMethodRefJoin {  le: LExpr; re: Expr | (le, re) ∈ set
                • (DistMethodRefJoin ({  fa: FieldAccess
                                         | le
                                           = FA fa
                                         • (let shareSet ==
                                                {  e1,
                                                   e2,
                                                   equale: LExpr
                                                   | (e1,
                                                      e2)
                                                     ∈ rel
                                                     ∧ FA (front fa)
                                                       = e1
                                                     ∧ equale
                                                       = MergeShareExprExprs (e2,
                                                                              e1,
                                                                              le)
                                                   • (equale,
                                                      re) };
                                            refSet ==
                                                {  e1,
                                                   e2,
                                                   equale: LExpr
                                                   | (e1,
                                                      e2)
                                                     ∈ rel
                                                     ∧ FA (front fa)
                                                       = e1
                                                     ∧ equale
                                                       = MergeShareExprExprs (e2,
                                                                              e1,
                                                                              le)
```

```
                                    • (equale,
                                       ref re) }
                                • AddAsgnFieldsMethodRefSet (shareSet,
                                                         rel,
                                                         (MethodRefUpdateSet (refSet,
                                                                         ref)))) }
                           ∪ {  ae: ArrayElement
                               | le
                                 = ID (arrayElement ae)
                              • (let shareSet ==
                                    {  e1,
                                      e2,
                                      equale: LExpr;
                                      v: Variable;
                                      fa: FieldAccess
                                       | (e1,
                                          e2)
                                         ∈ rel
                                        ∧ v.name
                                           = ae.name
                                        ∧ last fa
                                           = var v
                                        ∧ (e1
                                            = ID (var v)
                                          ∨ e1
                                             = FA fa)
                                        ∧ equale
                                           = MergeShareExprExprs (e2,
                                                               e1,
                                                               le)
                                     • (equale,
                                        re) };
                                 refSet ==
                                    {  e1,
                                      e2,
                                      equale: LExpr;
                                      v: Variable;
                                      fa: FieldAccess
                                       | (e1,
                                          e2)
                                         ∈ rel
                                        ∧ v.name
                                           = ae.name
                                        ∧ last fa
                                           = var v
                                        ∧ (e1
                                            = ID (var v)
                                          ∨ e1
                                             = FA fa)
                                        ∧ equale
                                           = MergeShareExprExprs (e2,
                                                               e1,
                                                               le)
                                     • (equale,
                                        ref re) }
                                 • AddAsgnFieldsMethodRefSet (shareSet,
                                                  rel,
                                                  (MethodRefUpdateSet (refSet,
                                                          ref)))) })) }
```

```
MethodRefAddPropertiesAsgn: LExpr × Expr × MethodProperties ⇸ MethodProperties
─────────────────
∀ properties: MethodProperties; le: LExpr; re: Expr
  • ∃ rel: ExprShareRelation; ref: MethodRefSet | properties = (rel, ref)
     • re ∈ dom ref
       ∧ MethodRefAddPropertiesAsgn (le, re, properties)
          = rel ↦ MethodRefUpdate (le, (ref re), ref)
       ∨ re ∉ dom ref
        ∧ MethodRefAddPropertiesAsgn (le, re, properties)
           = rel ↦ MethodRefUpdate (le, {(Erc re)}, ref)
```

```
CalcPropertiesAssignment: Method × MethodProperties × LExpr × Expr × LExpr ×
                    SCJmSafeProgram
                    ⇸ MethodProperties
─────────────────
 ∀ m: Method; properties: MethodProperties; le, cexpr: LExpr; re: Expr;
 p: SCJmSafeProgram
  • ∃ newle: LExpr; newre: Expr; v: Variable
      | newle = MergeExprs (cexpr, le, p)
      • re = Val
       ∧ CalcPropertiesAssignment (m, properties, le, re, cexpr, p)
          = properties
       ∨ re ≠ Val
        ∧ (re = This ∧ newre = cexpr
          ∨ re ≠ This ∧ newre = MergeExprs (cexpr, re, p))
        ∧ ((∃ v: Variable | le = ID (var v)
             • CalcPropertiesAssignment (m, properties, le, re,
                                cexpr, p)
               = AddAsgnFieldsProperties (newle, newre,
                            (MethodRefAddPropertiesAsgn (newle,
                                          newre,
                                          (ExprShareAddProperties (newle,
                                                   newre,
                                                   properties))))))
          ∨ (∃ fa: FieldAccess; ae: ArrayElement
             | le = FA fa ∨ le = ID (arrayElement ae)
             • CalcPropertiesAssignment (m, properties, le, re,
                                cexpr, p)
               = UpdateEqualExprsProperties (newle, newre,
                            (AddAsgnFieldsProperties (newle,
                                          newre,
                                          (MethodRefAddPropertiesAsgn (newle,
                                             newre,
                                              (ExprShareAddProperties (newle,
                                                     newre,
                                                     properties)))))))))
```

```
UpdateMethodRefSetForMrc: MethodRefSet × MetaRefCon ⇸ MethodRefSet
─────────────────
∀ ref: MethodRefSet; mrc: MetaRefCon
```

```
  • UpdateMethodRefSetForMrc (ref, mrc)
     = {  e: LExpr; mrcs: ℙ MetaRefCon | (e, mrcs) ∈ ref
          • e
           ↦ ∪ {  mrc1: mrcs • (AnalyseMetaRefCon (mrc1, mrc, ref)) } }
```

```
 ApplyMethodPropertiesProperties: Method × Method × MethodProperties ×
                          MethodProperties × ℙ LExpr × seq Expr ×
                          LExpr × LExpr × MetaRefCon × SCJmSafeProgram
                          ⇸ MethodProperties × ℙ LExpr
─────────────────────────
 ∀ m, method: Method; properties1, properties2: MethodProperties;
  localVars: ℙ LExpr; args: seq Expr; env: Env; cexpr, lexpr: LExpr;
  mrc: MetaRefCon; p: SCJmSafeProgram
   • ∃ rel, methRel: ExprShareRelation; ref, methRef: MethodRefSet
       | properties2 = (rel, ref) ∧ properties1 = (methRel, methRef)
       • let updatedShare ==
               UpdateMethodPropertiesCExprShare (methRel, args,
                                       method.visibleFields,
                                       cexpr, lexpr, p);
           updatedRef ==
                UpdateMethodPropertiesCExprRef (methRef, args,
                                       method.visibleFields,
                                       cexpr, lexpr, p)
          • ApplyMethodPropertiesProperties (m, method, properties1,
                                   properties2, localVars,
                                   args, cexpr, lexpr, mrc, p)
            = (UpdateEqualExprsPropertiesSet (updatedShare,
                                   (AddAsgnFieldsPropertiesSet (updatedShare,
                                        (ExprShareAddSet (updatedShare,
                                                        rel)
                                        ↦ MethodRefUpdateSet (
                                              (UpdateMethodRefSetForMrc (updatedRef,
                                                                    mrc)),
                                                        ref))))),
              localVars
              ∪ {  e: method.localVars • MergeExprs (cexpr, e, p) })
```

```
 ApplyPossibleMethodsProperties: Method × ℙ Method × seq Expr ×
                          MethodProperties × ℙ LExpr × LExpr × LExpr ×
                          MetaRefCon × SCJmSafeProgram
                          ⇸ MethodProperties × ℙ LExpr
─────────────────────────
 ∀ m: Method; methods: ℙ Method; args: seq Expr; properties: MethodProperties;
  cexpr, lexpr: LExpr; localVars: ℙ LExpr; mrc: MetaRefCon; p: SCJmSafeProgram
  • ∃ newLe: LExpr | newLe = MergeExprs (cexpr, lexpr, p)
      • methods = ∅
       ∧ ApplyPossibleMethodsProperties (m, methods, args, properties,
                               localVars, cexpr, lexpr, mrc, p)
          = (properties, localVars)
       ∨ methods ≠ ∅
        ∧ ApplyPossibleMethodsProperties (m, methods, args, properties,
                               localVars, cexpr, lexpr, mrc,
                               p)
           = (DistMethodPropertiesJoin {  m1: methods
                                  • (ApplyMethodPropertiesProperties (m,
                                                       m1,
                                                       (UpdateMethodPropertiesArgs (m1,
                                                                     args)),
                                                       properties,
                                                       localVars,
                                                       args,
                                                       cexpr,
                                                       lexpr,
                                                       mrc,
                                                       p)).1 },
          localVars
          ∪ ∪ {  m1: methods
              • (ApplyMethodPropertiesProperties (m, m1,
                                         (UpdateMethodPropertiesArgs (m1,
                                                       args)),
                                         properties,
                                         localVars,
                                         args, cexpr,
                                         lexpr, mrc,
                                         p)).2 })
```

```
CalcPropertiesNewInstance: Method × MethodProperties × ℙ LExpr × newInstance ×
            LExpr × MetaRefCon × SCJmSafeProgram
            ⇸ MethodProperties × ℙ LExpr
──────────────────────────
∀ m: Method; properties: MethodProperties; localVars: ℙ LExpr;
 nI: newInstance; cexpr: LExpr; mrc: MetaRefCon; p: SCJmSafeProgram
  • ∃ newLe, e1: LExpr; newMrc: MetaRefCon; constr: Method;
     rel: ExprShareRelation; ref: MethodRefSet
      | properties = (rel, ref)
       ∧ newLe = MergeExprs (cexpr, nI.le, p)
       ∧ (nI.mrc = Erc e1 ∧ newMrc = Erc (MergeExprs (cexpr, e1, p))
         ∨ nI.mrc ≠ Erc e1 ∧ newMrc = nI.mrc)
       ∧ constr = GetConstr (nI.type.type, nI.args, p)
     • CalcPropertiesNewInstance (m, properties, localVars, nI, cexpr,
                  mrc, p)
        = ApplyPossibleMethodsProperties (m, {constr}, nI.args,
                      (rel
                       ↦ MethodRefUpdate (newLe,
                             (AnalyseMetaRefCon (newMrc,
                                         mrc,
                                         ref)),
                             ref)),
                      localVars, cexpr, nI.le, mrc,
                      p)
```

```
CalcPropertiesMethod: Method × MethodProperties × ℙ LExpr × methodCall × LExpr
           × MetaRefCon × SCJmSafeProgram
           ⇸ MethodProperties × ℙ LExpr
──────────────────────────
∀ m: Method; properties: MethodProperties; localVars: ℙ LExpr; mc: methodCall;
 cexpr: LExpr; mrc: MetaRefCon; p: SCJmSafeProgram
  • CalcPropertiesMethod (m, properties, localVars, mc, cexpr, mrc, p)
     = ApplyPossibleMethodsProperties (m,
                    (GetMethodsFromSigs (mc.methods,
                            p)), mc.args,
                    properties, localVars, cexpr, mc.le,
                    mrc, p)
```

```
CalcPropertiesGetMemArea: Method × MethodProperties × getMemoryArea × LExpr ×
           MetaRefCon × SCJmSafeProgram
           ⇸ MethodProperties
────────────────
∀ m: Method; properties: MethodProperties; gma: getMemoryArea; cexpr: LExpr;
 mrc: MetaRefCon; p: SCJmSafeProgram
  • ∃ rel: ExprShareRelation; ref: MethodRefSet | properties = (rel, ref)
     • ∃ newRef, newExpr: LExpr; erc: MetaRefCon
         | newRef = MergeExprs (cexpr, gma.ref, p)
          ∧ (gma.e = This ∧ newExpr = cexpr
            ∨ gma.e ≠ This ∧ newExpr = MergeExprs (cexpr, gma.e, p))
          ∧ erc = Erc newExpr
         • CalcPropertiesGetMemArea (m, properties, gma, cexpr, mrc, p)
            = rel ↦ MethodRefUpdate (newRef, {erc}, ref)
```

```
CalcPropertiesCom: Method × MethodProperties × ℙ LExpr × Com × LExpr ×
         MetaRefCon × SCJmSafeProgram
         ⇸ MethodProperties × ℙ LExpr
────────────────
∀ m: Method; properties: MethodProperties; localVars: ℙ LExpr; c: Com;
 cexpr: LExpr; mrc: MetaRefCon; p: SCJmSafeProgram
  • c = Skip
    ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc, p)
       = (properties, localVars)
    ∨ (∃ d: Dec
        • c = Decl d
          ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                      p)
            = (AddDecToMethodProperties (properties, d),
              localVars ∪ {MergeExprs (cexpr, (ID (var d.var)), p)}))
    ∨ (∃ nI: newInstance
        • c = NewInstance nI
          ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                      p)
            = CalcPropertiesNewInstance (m, properties, localVars, nI,
                            cexpr, mrc, p))
    ∨ (∃ c1: Com
        • c = Scope c1
          ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                      p)
            = CalcPropertiesCom (m, properties, localVars, c1, cexpr,
                         mrc, p))
    ∨ (∃ le: LExpr; re: Expr
        • c = Asgn (le, re)
          ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                      p)
            = (CalcPropertiesAssignment (m, properties, le, re, cexpr,
                            p), localVars))
    ∨ (∃ c1, c2: Com
    • c = Seq (c1, c2)
     ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                 p)
        = CalcPropertiesCom (m,
                    (CalcPropertiesCom (m, properties,
                              localVars, c1,
                              cexpr, mrc, p)).1,
                    (CalcPropertiesCom (m, properties,
                              localVars, c1,
                              cexpr, mrc, p)).2,
                    c2, cexpr, mrc, p))
∨ (∃ e: Expr; c1, c2: Com | c = If (e, c1, c2)
    • (let trueResult ==
              CalcPropertiesCom (m, properties, localVars, c1,
                       cexpr, mrc, p);
        falseResult ==
              CalcPropertiesCom (m, properties, localVars, c2,
                       cexpr, mrc, p)
        • CalcPropertiesCom (m, properties, localVars, c, cexpr,
                   mrc, p)
          = (MethodPropertiesJoin (trueResult.1, falseResult.1),
            localVars ∪ trueResult.2 ∪ falseResult.2)))
∨ (∃ e: Expr; comSeq: seq Com
    • c = Switch (e, comSeq)
     ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                 p)
        = (DistMethodPropertiesJoin {   c: ran comSeq
                              • (CalcPropertiesCom (m,
                                      properties,
                                      localVars,
                                      c,
                                      cexpr,
                                      mrc,
                                      p)).1 },
          localVars
          ∪ ⋃ {   c: ran comSeq
               • (CalcPropertiesCom (m, properties, localVars,
                           c, cexpr, mrc, p)).2 }))
∨ (∃ c1, c2, c3: Com; exp: Expr
    • c = For (c1, exp, c2, c3)
     ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                 p)
        = ((CalcPropertiesCom (m,
                     (CalcPropertiesCom (m, properties,
                               localVars, c1,
                               cexpr, mrc,
                               p)).1,
                    localVars, (Seq (c2, c3)), cexpr,
                    mrc, p)).1,
          (CalcPropertiesCom (m, properties, localVars, c1, cexpr,
                    mrc, p)).2
          ∪ (CalcPropertiesCom (m,
                     (CalcPropertiesCom (m, properties,
                               localVars, c1,
                               cexpr, mrc,
                               p)).1,
                                 localVars, (Seq (c2, c3)), cexpr,
                                 mrc, p)).2))
     ∨ (∃ mc: methodCall
          • c = MethodCall mc
            ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                              p)
              = CalcPropertiesMethod (m, properties, localVars, mc,
                                 cexpr, mrc, p))
     ∨ (∃ mc: methodCall
          • c = EnterPrivateMemory mc
            ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                              p)
              = CalcPropertiesMethod (m, properties, localVars, mc,
                                 cexpr, (LowerMRC mrc), p))
     ∨ (∃ mrc2: MetaRefCon; mc: methodCall
          • c = ExecuteInAreaOf (mrc2, mc)
            ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                              p)
              = CalcPropertiesMethod (m, properties, localVars, mc,
                                 cexpr, mrc2, p))
     ∨ (∃ mc: methodCall
          • c = ExecuteInOuterArea mc
            ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                              p)
              = CalcPropertiesMethod (m, properties, localVars, mc,
                                 cexpr, (RaiseMRC mrc), p))
     ∨ (∃ gma: getMemoryArea
          • c = GetMemoryArea gma
            ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                              p)
              = (CalcPropertiesGetMemArea (m, properties, gma, cexpr,
                                   mrc, p), localVars))
     ∨ (∃ c1, c2: Com; eseq: seq Expr; comseq: seq Com
          • c = Try (c1, eseq, comseq, c2)
            ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                              p)
              = (MethodPropertiesJoin ((MethodPropertiesJoin ((CalcPropertiesCom (m,
                                                   properties,
                                                   localVars,
                                                   c1,
                                                   cexpr,
                                                   mrc,
                                                   p)).1,
                                        (DistMethodPropertiesJoin {   com: ran comseq
                                                     • (CalcPropertiesCom (m,
                                                               properties,
                                                               localVars,
                                                               com,
                                                               cexpr,
                                                               mrc,
                                                               p)).1  }))),
                                 (CalcPropertiesCom (m, properties,
                                           localVars, c2,
                                           cexpr, mrc,
                                           p)).1),
                  (CalcPropertiesCom (m, properties, localVars, c1, cexpr,
                    mrc, p)).2
          ∪ ⋃ {   com: ran comseq
                 • (CalcPropertiesCom (m, properties, localVars,
                           com, cexpr, mrc, p)).2  }
          ∪ (CalcPropertiesCom (m, properties, localVars, c2,
                      cexpr, mrc, p)).2))
   ∨ (∃ e: Expr; c1: Com
      • c = While (e, c1)
       ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                  p)
          = CalcPropertiesCom (m, properties, localVars, c1, cexpr,
                    mrc, p))
   ∨ (∃ c1: Com; e: Expr
      • c = DoWhile (c1, e)
       ∧ CalcPropertiesCom (m, properties, localVars, c, cexpr, mrc,
                  p)
          = CalcPropertiesCom (m, properties, localVars, c1, cexpr,
                    mrc, p))
```

```
GetEmbeddedInClasses: ℙ Name × SCJmSafeProgram ⇸ ℙ Name
────────────────
∀ names: ℙ Name; p: SCJmSafeProgram
  • ∃ names': ℙ Name
      | names'
         = names
          ∪ {   c: p.classes |  c.name ∈ names ∧ c.embeddedIn ≠ Empty
                • c.embeddedIn  }
      • # names' = # names ∧ GetEmbeddedInClasses (names, p) = names
       ∨ # names' > # names
        ∧ GetEmbeddedInClasses (names, p)
           = GetEmbeddedInClasses (names', p)
```

```
AnalyseMethodVisibleFields: Method × SCJmSafeProgram ⇸ ℙ LExpr
────────────────
∀ method: Method; p: SCJmSafeProgram
  • let classes  == GetEmbeddedInClasses ({method.class}, p)
    • AnalyseMethodVisibleFields (method, p)
       = {   d: ran p.safelet.fields |  p.safelet.name ∈ classes
             • ID (var d.var)  }
        ∪ {   d: ran p.missionSeq.fields |  p.missionSeq.name ∈ classes
              • ID (var d.var)  }
        ∪ ⋃ {   m: p.missions |  m.name ∈ classes
                • {   d: ran m.fields •  (ID (var d.var))  }  }
        ∪ ⋃ {   h: p.handlers |  h.name ∈ classes
                • {   d: ran h.fields •  (ID (var d.var))  }  }
        ∪ ⋃ {   c: p.classes |  c.name ∈ classes
                • {   d: ran c.fields •  (ID (var d.var))  }  }
```

```
BuildMethodPropertiesMethod: Method × SCJmSafeProgram ⇸ Method
────────────────
∀ method: Method; p: SCJmSafeProgram
  • ∃ method': Method
      | method'.name = method.name
```

$$
\begin{array}{l}
\quad\quad \land method'.returnType = method.returnType \\
\quad\quad \land method'.type = method.type \\
\quad\quad \land method'.params = method.params \\
\quad\quad \land method'.class = method.class \\
\quad\quad \land method'.body = method.body \\
\quad\quad \land method'.localVars \\
\quad\quad\quad = (CalcPropertiesCom\ (method, (\emptyset, \emptyset), \emptyset, method.body, Null, \\
\quad\quad\quad\quad\quad Current, p)).2 \\
\quad\quad \land method'.visibleFields = AnalyseMethodVisibleFields\ (method, p) \\
\quad\quad \land method'.properties \\
\quad\quad\quad = (CalcPropertiesCom\ (method, (\emptyset, \emptyset), \emptyset, method.body, Null, \\
\quad\quad\quad\quad\quad Current, p)).1 \\
\quad \bullet BuildMethodPropertiesMethod\ (method, p) = method'
\end{array}
$$

$MethodDependencies == Method \leftrightarrow Method$

$$
\begin{array}{l}
BuildMethodPropertiesMethods\colon \mathrm{seq}\ Method \times SCJmSafeProgram \rightarrowtail\!\!\!\!\!\!\!\!\; \mathrm{seq}\ Method \\
\hline
\forall methods\colon \mathrm{seq}\ Method; p\colon SCJmSafeProgram \\
\quad \bullet \#\, methods > 1 \\
\quad\quad \land BuildMethodPropertiesMethods\ (methods, p) \\
\quad\quad\quad = \langle BuildMethodPropertiesMethod\ ((head\ methods), p)\rangle \\
\quad\quad\quad {}^\frown BuildMethodPropertiesMethods\ ((tail\ methods), p) \\
\quad\quad \lor \#\, methods = 1 \\
\quad\quad\quad \land BuildMethodPropertiesMethods\ (methods, p) \\
\quad\quad\quad\quad = \langle BuildMethodPropertiesMethod\ ((head\ methods), p)\rangle
\end{array}
$$

$$
\begin{array}{l}
SortMethods\colon \mathrm{seq}\ Method \times MethodDependencies \rightarrow \mathrm{seq}\ Method \\
\hline
\forall sequence\colon \mathrm{seq}\ Method; deps\colon MethodDependencies \\
\quad \bullet \#\, sequence > 1 \\
\quad\quad \land (\forall m\colon \mathrm{ran}\ (tail\ sequence) \\
\quad\quad\quad \bullet (head\ sequence \mapsto m \notin deps^{\,*} \\
\quad\quad\quad\quad \land SortMethods\ (sequence, deps) \\
\quad\quad\quad\quad\quad = \langle head\ sequence\rangle {}^\frown SortMethods\ ((tail\ sequence), deps))) \\
\quad\quad \lor (\exists m\colon \mathrm{ran}\ (tail\ sequence) \\
\quad\quad\quad \bullet head\ sequence \mapsto m \in deps^{\,*} \\
\quad\quad\quad\quad \land SortMethods\ (sequence, deps) \\
\quad\quad\quad\quad\quad = SortMethods\ ((tail\ sequence {}^\frown \langle head\ sequence\rangle), deps)) \\
\quad\quad \lor \#\, sequence = 1 \land SortMethods\ (sequence, deps) = \langle head\ sequence\rangle
\end{array}
$$

$$
\begin{array}{l}
BuildMethodPropertiesSafelet\colon Safelet \times \mathbb{P}\ Method \rightarrow Safelet \\
\hline
\forall s\colon Safelet; methods\colon \mathbb{P}\ Method \\
\quad \bullet \exists s'\colon Safelet \\
\quad\quad \mid s'.name = s.name \\
\quad\quad\quad \land s'.fields = s.fields \\
\quad\quad\quad \land s'.init = s.init \\
\quad\quad\quad \land s'.initializeApplication = s.initializeApplication \\
\quad\quad\quad \land s'.getSequencer = s.getSequencer \\
\quad\quad\quad \land s'.missionSeq = s.missionSeq \\
\quad\quad\quad \land s'.methods \\
\quad\quad\quad\quad = \{\ m\colon methods \mid m.class = s.name \land m.name \neq s.name \bullet m\ \}
\end{array}
$$

$$\begin{array}{l}
\quad\quad \land\ s'.constrs \\
\quad\quad\quad = \{\ m: methods \mid m.class = s.name \land m.name = s.name \bullet m\ \} \\
\quad \bullet\ BuildMethodPropertiesSafelet\ (s, methods) = s'
\end{array}$$

$$\begin{array}{l}
BuildMethodPropertiesMSeq: MissionSeq \times \mathbb{P}\ Method \rightarrowtail MissionSeq \\
\hline
\forall\ ms: MissionSeq; methods: \mathbb{P}\ Method \\
\quad \bullet\ \exists\ ms': MissionSeq \\
\quad\quad \mid\ ms'.name = ms.name \\
\quad\quad \land\ ms'.fields = ms.fields \\
\quad\quad \land\ ms'.init = ms.init \\
\quad\quad \land\ ms'.missions = ms.missions \\
\quad\quad \land\ ms'.getNextMission = ms.getNextMission \\
\quad\quad \land\ ms'.methods \\
\quad\quad\quad = \{\ m: methods \mid m.class = ms.name \land m.name \neq ms.name \bullet m\ \} \\
\quad\quad \land\ ms'.constrs \\
\quad\quad\quad = \{\ m: methods \mid m.class = ms.name \land m.name = ms.name \bullet m\ \} \\
\quad \bullet\ BuildMethodPropertiesMSeq\ (ms, methods) = ms'
\end{array}$$

$$\begin{array}{l}
BuildMethodPropertiesMission: Mission \times \mathbb{P}\ Method \rightarrowtail Mission \\
\hline
\forall\ m: Mission; methods: \mathbb{P}\ Method \\
\quad \bullet\ \exists\ m': Mission \\
\quad\quad \mid\ m'.name = m.name \\
\quad\quad \land\ m'.fields = m.fields \\
\quad\quad \land\ m'.init = m.init \\
\quad\quad \land\ m'.initialize = m.initialize \\
\quad\quad \land\ m'.handlers = m.handlers \\
\quad\quad \land\ m'.cleanUp = m.cleanUp \\
\quad\quad \land\ m'.methods \\
\quad\quad\quad = \{\ meth: methods \mid meth.class = m.name \land meth.name \neq m.name \\
\quad\quad\quad\quad \bullet\ meth\ \} \\
\quad\quad \land\ m'.constrs \\
\quad\quad\quad = \{\ meth: methods \mid meth.class = m.name \land meth.name = m.name \\
\quad\quad\quad\quad \bullet\ meth\ \} \\
\quad \bullet\ BuildMethodPropertiesMission\ (m, methods) = m'
\end{array}$$

$$\begin{array}{l}
BuildMethodPropertiesMissions: \mathbb{P}\ Mission \times \mathbb{P}\ Method \rightarrowtail \mathbb{P}\ Mission \\
\hline
\forall\ missions: \mathbb{P}\ Mission; methods: \mathbb{P}\ Method \\
\quad \bullet\ BuildMethodPropertiesMissions\ (missions, methods) \\
\quad\quad = \{\ m: missions \bullet BuildMethodPropertiesMission\ (m, methods)\ \}
\end{array}$$

$$\begin{array}{l}
BuildMethodPropertiesHandler: Handler \times \mathbb{P}\ Method \rightarrowtail Handler \\
\hline
\forall\ h: Handler; methods: \mathbb{P}\ Method \\
\quad \bullet\ \exists\ h': Handler \\
\quad\quad \mid\ h'.name = h.name \\
\quad\quad \land\ h'.fields = h.fields \\
\quad\quad \land\ h'.init = h.init \\
\quad\quad \land\ h'.hAe = h.hAe \\
\quad\quad \land\ h'.methods
\end{array}$$

```
        = {  m: methods | m.class = h.name ∧ m.name ≠ h.name • m }
      ∧ h'.constrs
        = {  m: methods | m.class = h.name ∧ m.name = h.name • m }
    • BuildMethodPropertiesHandler (h, methods) = h'
```

```
BuildMethodPropertiesHandlers: ℙ Handler × ℙ Method ⇸ ℙ Handler
─────────────────────────
∀ handlers: ℙ Handler; methods: ℙ Method
  • BuildMethodPropertiesHandlers (handlers, methods)
    = {  h: handlers • BuildMethodPropertiesHandler (h, methods) }
```

```
BuildMethodPropertiesClass: Class × ℙ Method ⇸ Class
─────────────────────────
∀ c: Class; methods: ℙ Method
  • ∃ c': Class
      | c'.name = c.name
      ∧ c'.fields = c.fields
      ∧ c'.init = c.init
      ∧ c'.methods
        = {  m: methods | m.class = c.name ∧ m.name ≠ c.name • m }
      ∧ c'.constrs
        = {  m: methods | m.class = c.name ∧ m.name = c.name • m }
    • BuildMethodPropertiesClass (c, methods) = c'
```

```
BuildMethodPropertiesClasses: ℙ Class × ℙ Method ⇸ ℙ Class
─────────────────────────
∀ classes: ℙ Class; methods: ℙ Method
  • BuildMethodPropertiesClasses (classes, methods)
    = {  c: classes • BuildMethodPropertiesClass (c, methods) }
```

```
BuildMethodProperties: SCJmSafeProgram × MethodDependencies ⇸ SCJmSafeProgram
─────────────────────────
∀ p: SCJmSafeProgram; deps: MethodDependencies
  • let methods  == p.safelet.methods ∪ p.missionSeq.methods
              ∪ ⋃ {  m: p.missions • m.methods }
              ∪ ⋃ {  h: p.handlers • h.methods }
              ∪ ⋃ {  c: p.classes • c.methods }
    • ∃ methodSeq: seq Method; analysedMethods: ℙ Method;
      p': SCJmSafeProgram
        | ran methodSeq = methods
        ∧ # methodSeq = # methods
        ∧ analysedMethods
          = ran (BuildMethodPropertiesMethods ((SortMethods (methodSeq,
                                                  deps)),
                                            p))
      • p'.static = p.static
        ∧ p'.sInit = p.sInit
        ∧ p'.safelet
          = BuildMethodPropertiesSafelet (p.safelet,
                                    analysedMethods)
        ∧ p'.missionSeq
          = BuildMethodPropertiesMSeq (p.missionSeq,
```

$\qquad\qquad analysedMethods)$
$\quad \land\ p'.missions$
$\qquad = BuildMethodPropertiesMissions\ (p.missions,$
$\qquad\qquad analysedMethods)$
$\quad \land\ p'.handlers$
$\qquad = BuildMethodPropertiesHandlers\ (p.handlers,$
$\qquad\qquad analysedMethods)$
$\quad \land\ p'.classes$
$\qquad = BuildMethodPropertiesClasses\ (p.classes,$
$\qquad\qquad analysedMethods)$
$\quad \land\ BuildMethodProperties\ (p, deps) = p'$

$Dominates: RefCon \leftrightarrow RefCon$

$Dominates$
$\quad = \{(Prim \mapsto IMem), (IMem \mapsto MMem), (MMem \mapsto TPMMem\ 0)\}$
$\quad \cup \{\ x: \mathbb{N} \bullet\ (TPMMem\ x \mapsto TPMMem\ (x + 1))\ \}$
$\quad \cup \{\ h: Name \bullet\ (MMem \mapsto PRMem\ h)\ \}$
$\quad \cup \{\ h: Name \bullet\ (PRMem\ h \mapsto TPMem\ (h, 0))\ \}$
$\quad \cup \{\ h: Name; x: \mathbb{N} \bullet\ (TPMem\ (h, x) \mapsto TPMem\ (h, (x + 1)))\ \}$

$Dominates\_top: \mathbb{P}_1\ RefCon \rightarrowtail\!\!\!\!\!\!\rightarrow RefCon$

$\forall\ rcs: \mathbb{P}_1\ RefCon$
$\quad \bullet\ Dominates\_top\ rcs \in rcs$
$\qquad \land\ (\forall\ rc\_others: RefCon \mid rc\_others \in rcs$
$\qquad\quad \bullet\ Dominates\_top\ rcs \mapsto rc\_others \in Dominates^{*})$

$Dominates\_least: \mathbb{P}_1\ RefCon \rightarrowtail\!\!\!\!\!\!\rightarrow RefCon$

$\forall\ rcs: \mathbb{P}_1\ RefCon$
$\quad \bullet\ Dominates\_least\ rcs \in rcs$
$\qquad \land\ (\forall\ rc\_others: RefCon \mid rc\_others \in rcs$
$\qquad\quad \bullet\ rc\_others \mapsto Dominates\_least\ rcs \in Dominates^{*})$

Violation

$com: Com$
$e1: LExpr$
$rc1: RefCon$
$rc2: RefCon$

$(rc2, rc1) \notin Dominates^{*}$

$MRCDominates: MetaRefCon \leftrightarrow MetaRefCon$

$MRCDominates$
$\quad = \{\ e: LExpr; mrc: MetaRefCon \bullet\ (Erc\ e \mapsto mrc)\ \}$
$\quad \cup \{\ x: \mathbb{N} \bullet\ (CurrentPlus\ x \mapsto CurrentPlus\ (x - 1))\ \}$
$\quad \cup \{(CurrentPlus\ 0 \mapsto Current)\}$
$\quad \cup \{(Current \mapsto CurrentPrivate\ 0)\}$

$$
\begin{array}{l}
\quad \cup \{\ x: \mathbb{N} \bullet (CurrentPrivate\ x \mapsto CurrentPrivate\ (x + 1))\ \} \\
\quad \cup \{\ rcs1, rcs2: \mathbb{P}\ RefCon \\
\qquad \mid Dominates\_least\ rcs1 \mapsto Dominates\_top\ rcs2 \in Dominates^{*} \\
\qquad \bullet (Rcs\ rcs1 \mapsto Rcs\ rcs2)\ \}
\end{array}
$$

PropertiesViolation

$$
\begin{array}{l}
com: Com \\
e1: LExpr \\
mrc1: MetaRefCon \\
mrc2: MetaRefCon \\
\hline
(mrc2, mrc1) \notin MRCDominates^{*}
\end{array}
$$

$$
\begin{array}{l}
mSafeEnvStatic: Env \times Com \times SCJmSafeProgram \rightarrow\!\!\!\!\!\shortmid\;\; \mathbb{P}\ Violation \\
\hline
\forall env: Env; com: Com; p: SCJmSafeProgram \\
\quad \bullet \exists rel: ExprShareRelation; ref: ExprRefSet \mid env = (rel, ref) \\
\qquad \bullet mSafeEnvStatic\ (env, com, p) \\
\qquad\quad = \{\ e1: \mathrm{dom}\ ref; v: Violation \\
\qquad\qquad \mid e1 \in GetStaticVars\ p \\
\qquad\qquad\quad \wedge (Dominates\_least\ (ref\ e1), IMem) \notin Dominates^{*} \\
\qquad\qquad\quad \wedge v.com = com \\
\qquad\qquad\quad \wedge v.rc2 = IMem \\
\qquad\qquad\quad \wedge v.e1 = e1 \\
\qquad\qquad\quad \wedge v.rc1 = Dominates\_least\ (ref\ e1) \bullet v\ \}
\end{array}
$$

$$
\begin{array}{l}
mSafeEnvFields: Env \times Com \times \mathbb{P}\ LExpr \rightarrow\!\!\!\!\!\shortmid\;\; \mathbb{P}\ Violation \\
\hline
\forall env: Env; com: Com; localVars: \mathbb{P}\ LExpr \\
\quad \bullet \exists rel: ExprShareRelation; ref: ExprRefSet \mid env = (rel, ref) \\
\qquad \bullet mSafeEnvFields\ (env, com, localVars) \\
\qquad\quad = \{\ e1, e2: \mathrm{dom}\ ref; v: Violation \\
\qquad\qquad \mid FieldOf\ (e1, e2) = True \\
\qquad\qquad\quad \wedge e2 \notin localVars \\
\qquad\qquad\quad \wedge (Dominates\_least\ (ref\ e1), Dominates\_top\ (ref\ e2)) \\
\qquad\qquad\qquad \notin Dominates^{*} \\
\qquad\qquad\quad \wedge v.com = com \\
\qquad\qquad\quad \wedge v.e1 = e2 \\
\qquad\qquad\quad \wedge v.rc1 = Dominates\_least\ (ref\ e1) \\
\qquad\qquad\quad \wedge v.rc2 = Dominates\_top\ (ref\ e2) \bullet v\ \}
\end{array}
$$

$$
\begin{array}{l}
LongestPrefixOf: Env \times LExpr \rightarrow\!\!\!\!\!\shortmid\;\; LExpr \\
\hline
\forall env: Env; lexpr: LExpr \\
\quad \bullet \exists rel: ExprShareRelation; ref: ExprRefSet \mid env = (rel, ref) \\
\qquad \bullet \exists e: \mathrm{dom}\ ref \mid PrefixOf\ (e, lexpr) = True \\
\qquad\quad \bullet \forall e1: \mathrm{dom}\ ref \\
\qquad\qquad \mid e1 \neq e \wedge e1 \neq lexpr \wedge LengthOf\ e1 < LengthOf\ e \\
\qquad\qquad \bullet LongestPrefixOf\ (env, lexpr) = e
\end{array}
$$

```
$mSafeEnvIncomplete: Env \times Com \times \mathbb{P}\ LExpr \times SCJmSafeProgram \rightarrowtail \mathbb{P}\ Violation$
――――――――――――――――――――
$\forall\ env: Env; com: Com; localVars: \mathbb{P}\ LExpr; p: SCJmSafeProgram$
  $\bullet\ \exists\ rel: ExprShareRelation; ref: ExprRefSet \mid env = (rel, ref)$
    $\bullet$ let $uncheckedExprs ==$
          $\mathrm{dom}\ ref \setminus localVars \cup GetStaticVars\ p$
          $\cup \bigcup \{\ e: \mathrm{dom}\ ref$
              $\bullet\ \{\ e1: \mathrm{dom}\ ref$
                  $\mid e \neq e1 \wedge FieldOf\,(e, e1) = True \bullet\ e\ \}\ \}$
      $\bullet\ mSafeEnvIncomplete\,(env, com, localVars, p)$
        $= \{\ e1: uncheckedExprs; e2: LExpr; v: Violation$
            $\mid e2 = LongestPrefixOf\,(env, e1)$
             $\wedge\ (Dominates\_least\,(ref\ e1),$
               $Dominates\_top\,(ref\ e2)) \notin Dominates^{*}$
             $\wedge\ v.com = com$
             $\wedge\ v.e1 = e1$
             $\wedge\ v.rc1 = Dominates\_least\,(ref\ e1)$
             $\wedge\ v.rc2 = Dominates\_top\,(ref\ e2) \bullet\ v\ \}$
```

```
$mSafeEnvLocal: Env \times Com \times \mathbb{P}\ LExpr \times RefCon \rightarrowtail \mathbb{P}\ Violation$
――――――――――――――――――――
$\forall\ env: Env; com: Com; localVars: \mathbb{P}\ LExpr; rc: RefCon$
  $\bullet\ \exists\ rel: ExprShareRelation; ref: ExprRefSet \mid env = (rel, ref)$
    $\bullet\ mSafeEnvLocal\,(env, com, localVars, rc)$
      $= \{\ e1, e2: \mathrm{dom}\ ref; v: Violation$
          $\mid PrefixOf\,(e1, e2) = True$
           $\wedge\ e2 \in localVars$
           $\wedge\ (Dominates\_least\,(ref\ e1), rc) \notin Dominates^{*}$
           $\wedge\ v.com = com$
           $\wedge\ v.e1 = e2$
           $\wedge\ v.rc1 = Dominates\_least\,(ref\ e1)$
           $\wedge\ v.rc2 = rc \bullet\ v\ \}$
```

```
$mSafePropertiesLocal: MethodProperties \times Com \times \mathbb{P}\ LExpr \times MetaRefCon$
          $\rightarrowtail \mathbb{P}\ PropertiesViolation$
――――――――――――――――――――
$\forall\ properties: MethodProperties; com: Com; vars: \mathbb{P}\ LExpr; mrc: MetaRefCon$
  $\bullet\ \exists\ rel: ExprShareRelation; ref: MethodRefSet \mid properties = (rel, ref)$
    $\bullet\ mSafePropertiesLocal\,(properties, com, vars, mrc)$
      $= \bigcup \{\ e1, e2: \mathrm{dom}\ ref \mid PrefixOf\,(e1, e2) = True$
          $\bullet\ \{\ v: PropertiesViolation; mrc1: MetaRefCon;$
              $e: LExpr; rcs: \mathbb{P}\ RefCon$
              $\mid mrc1 \in ref\ e1$
               $\wedge\ mrc1 \neq Erc\ e$
               $\wedge\ mrc1 \neq Rcs\ rcs$
               $\wedge\ e2 \in vars$
               $\wedge\ (mrc1, mrc) \notin MRCDominates^{*}$
               $\wedge\ v.com = com$
               $\wedge\ v.e1 = e2$
               $\wedge\ v.mrc1 = mrc1$
               $\wedge\ v.mrc2 = mrc \bullet\ v\ \}\ \}$
```

```
$mSafePropertiesFields: Method \times MethodProperties \times Com \rightarrowtail \mathbb{P}\ PropertiesViolation$
```

$$\begin{array}{l}
\hline
\forall\ m: Method;\ properties: MethodProperties;\ com: Com \\
\quad \bullet\ \exists\ rel: ExprShareRelation;\ ref: MethodRefSet \mid properties = (rel, ref) \\
\qquad \bullet\ mSafePropertiesFields\ (m, properties, com) \\
\qquad\quad = \bigcup \{\ e1, e2: \mathrm{dom}\ ref \\
\qquad\qquad \mid FieldOf\ (e1, e2) = True \land e2 \notin m.localVars \\
\qquad\qquad \bullet \{\ v: PropertiesViolation;\ mrc1, mrc2: MetaRefCon; \\
\qquad\qquad\quad e: LExpr;\ rcs: \mathbb{P}\ RefCon \\
\qquad\qquad\quad \mid mrc1 \in ref\ e1 \\
\qquad\qquad\qquad \land\ mrc2 \in ref\ e2 \\
\qquad\qquad\qquad \land\ mrc1 \neq Erc\ e \\
\qquad\qquad\qquad \land\ mrc2 \neq Erc\ e \\
\qquad\qquad\qquad \land\ mrc1 \neq Rcs\ rcs \\
\qquad\qquad\qquad \land\ mrc2 \neq Rcs\ rcs \\
\qquad\qquad\qquad \land\ (mrc2, mrc1) \notin MRCDominates^{*} \\
\qquad\qquad\qquad \land\ v.com = com \\
\qquad\qquad\qquad \land\ v.e1 = e2 \\
\qquad\qquad\qquad \land\ v.mrc1 = mrc2 \\
\qquad\qquad\qquad \land\ v.mrc2 = mrc1 \bullet\ v\ \}\ \}
\end{array}$$

$$\begin{array}{l}
mSafePropertiesMethod: Method \times MethodProperties \times Method \times methodCall \times LExpr \\
\qquad\qquad \times MetaRefCon \times SCJmSafeProgram \\
\qquad\qquad \rightarrow\!\!\!\!\!+\ \mathbb{P}\ PropertiesViolation \\
\hline
\forall\ properties: MethodProperties;\ m, meth: Method;\ mc: methodCall;\ cexpr: LExpr; \\
\quad mrc: MetaRefCon;\ p: SCJmSafeProgram \\
\quad \bullet\ \exists\ properties': MethodProperties \\
\qquad \mid properties' \\
\qquad\quad = (ApplyPossibleMethodsProperties\ (m, \{meth\}, mc.args, \\
\qquad\qquad\qquad\qquad properties, \emptyset, cexpr, mc.le, \\
\qquad\qquad\qquad\qquad mrc, p)).1 \\
\qquad \bullet\ mSafePropertiesMethod\ (m, properties, meth, mc, cexpr, mrc, p) \\
\qquad\quad = mSafePropertiesFields\ (m, properties', (MethodCall\ mc)) \\
\qquad\quad \cup\ mSafePropertiesLocal\ (properties', (MethodCall\ mc), \\
\qquad\qquad\qquad meth.localVars, mrc)
\end{array}$$

$$\begin{array}{l}
mSafePropertiesMethodCall: Method \times MethodProperties \times methodCall \times LExpr \times \\
\qquad\qquad MetaRefCon \times SCJmSafeProgram \\
\qquad\qquad \rightarrow\!\!\!\!\!+\ \mathbb{P}\ PropertiesViolation \\
\hline
\forall\ m: Method;\ properties: MethodProperties;\ mc: methodCall;\ cexpr: LExpr; \\
\quad mrc: MetaRefCon;\ p: SCJmSafeProgram \\
\quad \bullet\ mSafePropertiesMethodCall\ (m, properties, mc, cexpr, mrc, p) \\
\qquad = \bigcup \{\ meth: GetMethodsFromSigs\ (mc.methods, p) \\
\qquad\quad \bullet\ (mSafePropertiesMethod\ (m, properties, meth, mc, cexpr, \\
\qquad\qquad\qquad mrc, p))\ \}
\end{array}$$

$$\begin{array}{l}
mSafeProperties: MethodProperties \times Method \times Com \times MetaRefCon \\
\qquad \rightarrow\!\!\!\!\!+\ \mathbb{P}\ PropertiesViolation \\
\hline
\forall\ properties: MethodProperties;\ m: Method;\ com: Com;\ mrc: MetaRefCon \\
\quad \bullet\ mSafeProperties\ (properties, m, com, mrc) \\
\qquad = mSafePropertiesFields\ (m, properties, com)
\end{array}$$

```
    ∪ mSafePropertiesLocal (properties, com, m.localVars, mrc)
```

```
mSafePropertiesCom: Method × MethodProperties × Com × LExpr × MetaRefCon ×
        SCJmSafeProgram
        ⇸ ℙ PropertiesViolation
─────────────────────
∀ m: Method; properties: MethodProperties; c: Com; cexpr: LExpr;
 mrc: MetaRefCon; p: SCJmSafeProgram
  • ∃ properties': MethodProperties
     | properties'
       = (CalcPropertiesCom (m, properties, ∅, c, cexpr, mrc, p)).1
    • c = Skip
      ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p) = ∅
      ∨ (∃ d: Dec
          • c = Decl d
           ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
             = ∅)
      ∨ (∃ nI: newInstance
          • c = NewInstance nI
           ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
             = mSafeProperties (properties', m, c, mrc))
      ∨ (∃ c1: Com | c = Scope c1
          • mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
            = mSafePropertiesCom (m, properties, c1, cexpr, mrc,
                        p))
      ∨ (∃ le: LExpr; re: Expr
          • c = Asgn (le, re)
           ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
             = mSafeProperties (properties', m, c, mrc))
      ∨ (∃ c1, c2: Com | c = Seq (c1, c2)
          • (let c1result ==
                 mSafePropertiesCom (m, properties, c1, cexpr,
                            mrc, p);
              c1properties ==
                (CalcPropertiesCom (m, properties, ∅, c1, cexpr,
                            mrc, p)).1
             • (let c2result ==
                    mSafePropertiesCom (m, c1properties, c2,
                              cexpr, mrc, p)
                 • mSafePropertiesCom (m, properties, c, cexpr,
                            mrc, p)
                   = c1result ∪ c2result)))
      ∨ (∃ e: Expr; c1, c2: Com | c = If (e, c1, c2)
          • (let c1result ==
                 mSafePropertiesCom (m, properties, c1, cexpr,
                            mrc, p);
              c2result ==
                 mSafePropertiesCom (m, properties, c2, cexpr,
                            mrc, p)
             • mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
               = c1result ∪ c2result))
      ∨ (∃ e: Expr; comSeq: seq Com | c = Switch (e, comSeq)
          • (let comresults ==
                 { c: ran comSeq
                    • mSafePropertiesCom (m, properties, c,
                                    cexpr, mrc, p) }
                  • mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                      = ∪ comresults))
       ∨ (∃ c1, c2, c3: Com; exp: Expr
            • c = For (c1, exp, c2, c3)
              ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesCom (m, properties, c1, cexpr, mrc,
                                 p)
                    ∪ mSafePropertiesCom (m, properties,
                                  (Seq (c2, c3)), cexpr, mrc,
                                  p))
       ∨ (∃ mc: methodCall
            • c = MethodCall mc
              ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesMethodCall (m, properties, mc,
                                      cexpr, mrc, p))
       ∨ (∃ mc: methodCall
            • c = EnterPrivateMemory mc
              ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesMethodCall (m, properties, mc,
                                      cexpr, mrc, p))
       ∨ (∃ mrc2: MetaRefCon; mc: methodCall
            • c = ExecuteInAreaOf (mrc2, mc)
              ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesMethodCall (m, properties, mc,
                                      cexpr, mrc, p))
       ∨ (∃ mc: methodCall
            • c = ExecuteInOuterArea mc
              ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesMethodCall (m, properties, mc,
                                      cexpr, mrc, p))
       ∨ (∃ gma: getMemoryArea
            • c = GetMemoryArea gma
              ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafeProperties (properties', m, c, mrc))
       ∨ (∃ c1, c2: Com; eseq: seq Expr; comseq: seq Com
            | c = Try (c1, eseq, comseq, c2)
            • (let c1result ==
                      mSafePropertiesCom (m, properties, c1, cexpr,
                                   mrc, p);
                 c1properties ==
                    (CalcPropertiesCom (m, properties, ∅, c1, cexpr,
                                  mrc, p)).1
                 • (let comSeqResult ==
                          { com: ran comseq
                              • mSafePropertiesCom (m, c1properties,
                                          com, cexpr, mrc,
                                          p) };
                      comSeqProperties ==
                        DistMethodPropertiesJoin { com: ran comseq
                                            • (CalcPropertiesCom (m,
                                                      c1properties,
                                                      ∅,
                                                      com,
                                                      cexpr,
                                                      mrc,
                                          p)).1 }
                    • (let c2result ==
                          mSafePropertiesCom (m,
                                    comSeqProperties,
                                    c2, cexpr, mrc, p)
                       • mSafePropertiesCom (m, properties, c,
                                   cexpr, mrc, p)
                         = c1result ∪ ∪ comSeqResult
                           ∪ c2result))))
          ∨ (∃ e: Expr; c1: Com
              • c = While (e, c1)
                ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesCom (m, properties, c1, cexpr, mrc,
                          p))
          ∨ (∃ e: Expr; c1: Com
              • c = DoWhile (c1, e)
                ∧ mSafePropertiesCom (m, properties, c, cexpr, mrc, p)
                  = mSafePropertiesCom (m, properties, c1, cexpr, mrc,
                          p))
```

```
mSafeMethodProperties: Method × SCJmSafeProgram ⇸ ℙ PropertiesViolation
────────────────
∀ m: Method; p: SCJmSafeProgram
  • mSafeMethodProperties (m, p)
    = mSafePropertiesCom (m, (∅, ∅), m.body, Null, Current, p)
```

```
mSafeEnv: Env × Com × ℙ LExpr × RefCon × SCJmSafeProgram ⇸ ℙ Violation
────────────────
∀ env: Env; com: Com; localVars: ℙ LExpr; rc: RefCon; p: SCJmSafeProgram
  • mSafeEnv (env, com, localVars, rc, p)
    = mSafeEnvStatic (env, com, p)
     ∪ mSafeEnvLocal (env, com, localVars, rc)
     ∪ mSafeEnvFields (env, com, localVars)
     ∪ mSafeEnvIncomplete (env, com, localVars, p)
```

```
mSafeMethod: Env × Method × methodCall × ℙ LExpr × LExpr × RefCon × RefCon ×
        SCJmSafeProgram
        ⇸ ℙ Violation
────────────────
∀ env: Env; m: Method; mc: methodCall; localVars: ℙ LExpr; cexpr: LExpr; rc,
  currentrc: RefCon; p: SCJmSafeProgram
  • ∃ env’: Env
     | env’
        = ApplyPossibleMethods ({m}, mc.args, env, cexpr, mc.le, rc, p)
     • mSafeMethod (env, m, mc, localVars, cexpr, rc, currentrc, p)
        = mSafeEnv (env’, (MethodCall mc), (m.localVars ∪ localVars),
                currentrc, p)
```

```
mSafeMethodCall: Env × methodCall × ℙ LExpr × LExpr × RefCon × RefCon ×
          SCJmSafeProgram
          ⇸ ℙ Violation
────────────────
```

```
∀ env: Env; mc: methodCall; localVars: ℙ LExpr; cexpr: LExpr; rc,
  currentrc: RefCon; p: SCJmSafeProgram
   • mSafeMethodCall (env, mc, localVars, cexpr, rc, currentrc, p)
       = ∪ {  m: GetMethodsFromSigs (mc.methods, p)
             • (mSafeMethod (env, m, mc, localVars, cexpr, rc, currentrc,
                        p)) }
```

```
mSafeCom: Env × Com × ℙ LExpr × LExpr × RefCon × SCJmSafeProgram ⇸ ℙ Violation
─────────────────────────────
∀ env: Env; c: Com; localVars: ℙ LExpr; cexpr: LExpr; rc: RefCon;
  p: SCJmSafeProgram
   • ∃ env': Env | env' = CalcEnvCom (env, c, cexpr, rc, p)
       • c = Skip ∧ mSafeCom (env, c, localVars, cexpr, rc, p) = ∅
         ∨ (∃ d: Dec
               • c = Decl d
                 ∧ mSafeCom (env, c, localVars, cexpr, rc, p) = ∅)
         ∨ (∃ nI: newInstance
               • c = NewInstance nI
                 ∧ mSafeCom (env, c, localVars, cexpr, rc, p)
                     = mSafeEnv (env', c, localVars, rc, p))
         ∨ (∃ c1: Com
               • c = Scope c1
                 ∧ mSafeCom (env, c, localVars, cexpr, rc, p)
                     = mSafeCom (env, c1, localVars, cexpr, rc, p))
         ∨ (∃ le: LExpr; re: Expr
               • c = Asgn (le, re)
                 ∧ mSafeCom (env, c, localVars, cexpr, rc, p)
                     = mSafeEnv (env', c, localVars, rc, p))
         ∨ (∃ c1, c2: Com | c = Seq (c1, c2)
               • (let c1result ==
                          mSafeCom (env, c1, localVars, cexpr, rc, p);
                    c1env == CalcEnvCom (env, c1, cexpr, rc, p)
                   • (let c2result ==
                            mSafeCom (c1env, c2, localVars, cexpr, rc,
                                  p)
                       • mSafeCom (env, c, localVars, cexpr, rc, p)
                          = c1result ∪ c2result)))
         ∨ (∃ e: Expr; c1, c2: Com | c = If (e, c1, c2)
               • (let c1result ==
                          mSafeCom (env, c1, localVars, cexpr, rc, p);
                    c2result ==
                           mSafeCom (env, c2, localVars, cexpr, rc, p)
                   • mSafeCom (env, c, localVars, cexpr, rc, p)
                      = c1result ∪ c2result))
         ∨ (∃ e: Expr; comSeq: seq Com | c = Switch (e, comSeq)
               • (let comresults ==
                       {  com: ran comSeq
                           • mSafeCom (env, com, localVars, cexpr, rc,
                                  p) }
                   • mSafeCom (env, c, localVars, cexpr, rc, p)
                      = ∪ comresults))
         ∨ (∃ c1, c2, c3: Com; exp: Expr
               • (let c1env == CalcEnvCom (env, c1, cexpr, rc, p)
                   • c = For (c1, exp, c2, c3)
                     ∧ mSafeCom (env, c, localVars, cexpr, rc, p)
```

```
                = $mSafeCom$ ($c1env$, $c1$, $localVars$, $cexpr$, $rc$, $p$)
                  $\cup$ $mSafeCom$ ($env$, ($Seq$ ($c2$, $c3$)), $localVars$,
                             $cexpr$, $rc$, $p$)))
$\vee$ ($\exists$ $mc$: $methodCall$
    $\bullet$ $c$ = $MethodCall$ $mc$
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
        = $mSafeMethodCall$ ($env$, $mc$, $localVars$, $cexpr$, $rc$, $rc$,
                  $p$))
$\vee$ ($\exists$ $mc$: $methodCall$
    $\bullet$ $c$ = $EnterPrivateMemory$ $mc$
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
        = $mSafeMethodCall$ ($env$, $mc$, $localVars$, $cexpr$,
                  ($LowerRC$ $rc$), $rc$, $p$))
$\vee$ ($\exists$ $mrc$: $MetaRefCon$; $mc$: $methodCall$; $ref$: $ExprRefSet$
    | $ref$ = $env$.2
    $\bullet$ $c$ = $ExecuteInAreaOf$ ($mrc$, $mc$)
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
        = $\bigcup$ { $rc1$: $RCsFromMRC$ ($mrc$, $rc$, $ref$, $cexpr$)
              $\bullet$ ($mSafeMethodCall$ ($env$, $mc$, $localVars$,
                        $cexpr$, $rc1$, $rc$, $p$)) })
$\vee$ ($\exists$ $mc$: $methodCall$
    $\bullet$ $c$ = $ExecuteInOuterArea$ $mc$
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
        = $mSafeMethodCall$ ($env$, $mc$, $localVars$, $cexpr$,
                  ($RaiseRC$ $rc$), $rc$, $p$))
$\vee$ ($\exists$ $gma$: $getMemoryArea$
    $\bullet$ $c$ = $GetMemoryArea$ $gma$
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
        = $mSafeEnv$ ($env'$, $c$, $localVars$, $rc$, $p$))
$\vee$ ($\exists$ $c1$, $c2$: $Com$; $eseq$: seq $Expr$; $comseq$: seq $Com$
    | $c$ = $Try$ ($c1$, $eseq$, $comseq$, $c2$)
    $\bullet$ (**let** $c1result$ ==
              $mSafeCom$ ($env$, $c1$, $localVars$, $cexpr$, $rc$, $p$);
         $c1env$ == $CalcEnvCom$ ($env$, $c1$, $cexpr$, $rc$, $p$)
      $\bullet$ (**let** $comSeqresult$ ==
                { $com$: ran $comseq$
                    $\bullet$ $mSafeCom$ ($c1env$, $com$, $localVars$,
                            $cexpr$, $rc$, $p$) };
            $comSeqEnv$ ==
              $DistEnvJoin$ { $com$: ran $comseq$
                          $\bullet$ ($CalcEnvCom$ ($c1env$, $com$,
                                  $cexpr$, $rc$,
                                  $p$)) }
        $\bullet$ (**let** $c2result$ ==
                  $mSafeCom$ ($comSeqEnv$, $c2$, $localVars$,
                         $cexpr$, $rc$, $p$)
          $\bullet$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
              = $c1result$ $\cup$ $\bigcup$ $comSeqresult$
               $\cup$ $c2result$))))
$\vee$ ($\exists$ $e$: $Expr$; $c1$: $Com$
    $\bullet$ $c$ = $While$ ($e$, $c1$)
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
        = $mSafeCom$ ($env$, $c1$, $localVars$, $cexpr$, $rc$, $p$))
$\vee$ ($\exists$ $e$: $Expr$; $c1$: $Com$
    $\bullet$ $c$ = $DoWhile$ ($c1$, $e$)
      $\wedge$ $mSafeCom$ ($env$, $c$, $localVars$, $cexpr$, $rc$, $p$)
```

$= mSafeCom\ (env, c1, localVars, cexpr, rc, p))$

$mSafeHandler: Env \times Handler \times LExpr \times SCJmSafeProgram \nrightarrow \mathbb{P}\ Violation$

$\forall\ env: Env; h: Handler; cexpr: LExpr; p: SCJmSafeProgram$
$\quad \bullet\ mSafeHandler\ (env, h, cexpr, p)$
$\qquad = mSafeCom\ (env, h.hAe, (LocalVars\ h.hAe), cexpr, (PRMem\ h.name), p)$

$mSafeHandlers: Env \times Mission \times \mathbb{P}\ Handler \times SCJmSafeProgram \nrightarrow \mathbb{P}\ Violation$

$\forall\ env: Env; m: Mission; handlers: \mathbb{P}\ Handler; p: SCJmSafeProgram$
$\quad \bullet\ \exists\ h: Handler \mid h \in handlers$
$\qquad \bullet\ mSafeHandlers\ (env, m, handlers, p)$
$\qquad\quad = mSafeHandler\ (env, h, (GetHandlerExpr\ (p, h, m)), p)$
$\qquad\quad \cup\ mSafeHandlers\ ((CalcEnvHandler\ (env, h,$
$\qquad\qquad\qquad (GetHandlerExpr\ (p, h, m)),$
$\qquad\qquad\qquad p)), m, (handlers \setminus \{h\}), p)$

$mSafeMission: Env \times Mission \times SCJmSafeProgram \nrightarrow \mathbb{P}\ Violation$

$\forall\ env: Env; m: Mission; p: SCJmSafeProgram$
$\quad \bullet\ \mathbf{let}\ initializeResult == $
$\qquad\qquad mSafeCom\ (env, m.initialize, (LocalVars\ m.initialize),$
$\qquad\qquad\quad (GetMissionExpr\ (p, m)), MMem, p);$
$\qquad initializeEnv ==$
$\qquad\quad CalcEnvCom\ (env, m.initialize, (GetMissionExpr\ (p, m)), MMem, p)$
$\qquad \bullet\ \mathbf{let}\ handlersResult ==$
$\qquad\qquad mSafeHandlers\ ((RemoveExprSetEnv\ ((LocalVars\ m.initialize$
$\qquad\qquad\qquad \setminus GetHandlerExprs\ (p,$
$\qquad\qquad\qquad\quad m)),$
$\qquad\qquad\qquad initializeEnv)), m,$
$\qquad\qquad (GetHandlers\ (p, m.handlers)), p);$
$\qquad\quad handlersEnv ==$
$\qquad\qquad CalcEnvHandlers\ ((RemoveExprSetEnv\ ((LocalVars\ m.initialize$
$\qquad\qquad\qquad \setminus GetHandlerExprs\ (p,$
$\qquad\qquad\qquad\quad m)),$
$\qquad\qquad\qquad initializeEnv)), m,$
$\qquad\qquad (GetHandlers\ (p, m.handlers)), p)$
$\qquad\quad \bullet\ \mathbf{let}\ cleanUpResult ==$
$\qquad\qquad mSafeCom\ (handlersEnv, m.cleanUp, (LocalVars\ m.cleanUp),$
$\qquad\qquad\quad (GetMissionExpr\ (p, m)), MMem, p)$
$\qquad\qquad \bullet\ mSafeMission\ (env, m, p)$
$\qquad\qquad\quad = initializeResult \cup handlersResult \cup cleanUpResult$

$mSafeMissions: Env \times \mathbb{P}\ Mission \times SCJmSafeProgram \nrightarrow \mathbb{P}\ Violation$

$\forall\ env: Env; missions: \mathbb{P}\ Mission; p: SCJmSafeProgram$
$\quad \bullet\ \exists\ m: Mission \mid m \in missions$
$\qquad \bullet\ mSafeMissions\ (env, missions, p)$
$\qquad\quad = mSafeMission\ (env, m, p)$
$\qquad\quad \cup\ mSafeMissions\ ((CalcEnvMission\ (env, m,$
$\qquad\qquad\qquad (GetMissionExpr\ (p, m)),$

```
                    p)), (missions \ {m}), p)
```

```
mSafeMissionSeq: Env × MissionSeq × SCJmSafeProgram ⇸ ℙ Violation
─────────────────────
∀ env: Env; ms: MissionSeq; p: SCJmSafeProgram
  • let getNextMissionResult ==
          mSafeCom (env, ms.getNextMission, (LocalVars ms.getNextMission),
               (GetMissionSeqExpr (p, ms)), MMem, p);
      getNextMissionEnv ==
          CalcEnvCom (env, ms.getNextMission, (GetMissionSeqExpr (p, ms)),
               MMem, p)
    • let missionsResult ==
            mSafeMissions ((RemoveExprSetEnv ((LocalVars ms.getNextMission
                                  \ GetMissionExprs p),
                                 getNextMissionEnv)),
                   p.missions, p)
      • mSafeMissionSeq (env, ms, p)
          = getNextMissionResult ∪ missionsResult
```

```
mSafeSafelet: Env × Safelet × SCJmSafeProgram ⇸ ℙ Violation
─────────────────────
∀ env: Env; s: Safelet; p: SCJmSafeProgram
  • let initializeResult ==
            mSafeCom (env, s.initializeApplication,
                 (LocalVars s.initializeApplication), Null, IMem, p);
      initializeEnv ==
               CalcEnvCom (env, s.initializeApplication, Null, IMem, p)
    • let getSeqResult ==
            mSafeCom ((RemoveExprSetEnv ((LocalVars s.initializeApplication),
                              initializeEnv)),
                 s.getSequencer, (LocalVars s.getSequencer), Null,
                 IMem, p);
        getSeqEnv ==
             CalcEnvCom (initializeEnv, s.getSequencer, Null, IMem, p)
      • let seqResult ==
              mSafeMissionSeq ((RemoveExprSetEnv ((LocalVars s.getSequencer
                                   \ {GetMissionSeqExpr (p,
                                              p.missionSeq)}),
                                  getSeqEnv)),
                     p.missionSeq, p)
        • mSafeSafelet (env, s, p)
            = initializeResult ∪ getSeqResult ∪ seqResult
```

```
mSafeProgram: SCJProgram
       ⇸ SCJmSafeProgram × ℙ PropertiesViolation × ℙ Violation
─────────────────────
∀ scjProgram: SCJProgram
  • ∃ scjmsafe, scjmsafe': SCJmSafeProgram; deps: MethodDependencies;
    env: Env
      | scjmsafe = Translate scjProgram
       ∧ scjmsafe' = BuildMethodProperties (scjmsafe, deps)
       ∧ env
         = DistEnvJoin {  c: scjmsafe'.sInit
```

```
                    • (CalcEnvCom ((AddDecsToEnv ((∅, ∅),
                                   scjmsafe'.static)),
                          c, Null, IMem, scjmsafe'))  }
  • let methods  ==
        scjmsafe'.safelet.methods ∪ scjmsafe'.missionSeq.methods
        ∪ ⋃ {  m: scjmsafe'.missions •  m.methods  }
        ∪ ⋃ {  h: scjmsafe'.handlers •  h.methods  }
        ∪ ⋃ {  c: scjmsafe'.classes •  c.methods  }
    • mSafeProgram scjProgram
       = (scjmsafe',
        ⋃ {  m: methods
             • (mSafeMethodProperties (m, scjmsafe'))  },
        mSafeSafelet (env, scjmsafe'.safelet, scjmsafe'))
```

References

[1] Jamaica Virtual Machine. `https://www.aicas.com/cms/en/JamaicaVM`.

[2] Java Compiler Tree API. `http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/tree/package-summary.html`. Accessed: 2014-06-25.

[3] SCJmSafe Automatic Translation and Checking Tool. `http://www.cs.york.ac.uk/circus/hijac/tools.html`.

[4] T.J. Watson libraries for analysis (WALA). `http://wala.sf.net/`.

[5] W. Ahrendt, T. Baar, B. Beckert, W. Menzel, and P.H. Schmitt. The KeY tool, integrating object oriented design and formal verification. software and systems modeling 4, 2005.

[6] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[7] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. *ACM SIGPLAN Notices*, 38(5):324–337, May 2003.

[8] A. Burns. The ravenscar profile. *ACM SIGAda Ada Letters*, 19(4):49–52, 1999.

[9] A. Burns and A.J. Wellings. *Real-time systems and programming languages*, volume 2097. Addison-Wesley, 1998.

[10] A. Cavalcanti, A. Sampaio, and J. Woodcock. Unifying classes and processes. *Software & Systems Modeling*, 4(3):277–296, 2005.

[11] A. Cavalcanti, A.J. Wellings, and J. Woodcock. The Safety-Critical Java memory model: A formal account. *FM 2011: Formal Methods*, pages 246–261, 2011.

[12] A. Cavalcanti, A.J. Wellings, J. Woodcock, K. Wei, and F. Zeyda. Safety-Critical Java in Circus. In A.J. Wellings and A.P. Ravn, editors, *Java Technologies for Real-time and Embedded Systems*, pages 20–29. ACM, 2011.

[13] Z. Chen. *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[14] A.E. Dalsgaard, R.R. Hansen, and M. Schoeberl. Private memory allocation analysis for SCJ. In *Proceedings of Java Technologies for Real-time and Embedded Systems*, pages 9–17. ACM, 2012.

[15] C. Engel. Deductive verification of RTSJ programs. In *Proceedings of the 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC 2008)*, 2008.

[16] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM, 2002.

[17] W. Harwood, A. Cavalcanti, and J. Woodcock. A theory of pointers for the UTP. In *Theoretical Aspects of Computing-ICTAC 2008*, pages 141–155. Springer, 2008.

[18] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[19] M.T. Higuera-Toledano and M.A. de Miguel-Cabello. Dynamic detection of access errors and illegal references in RTSJ. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pages 101–110. IEEE, 2002.

[20] C. A. R. Hoare. Unified Theories of Programming. Technical report, Oxford University Computing Laboratory, Oxford - UK, 1994.

[21] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CDx: a family of real-time Java benchmarks. In *Proceedings of Java Technologies for Real-time and Embedded Systems*, pages 41–50. ACM, 2009.

[22] T. Kalibera, P. Parizek, M. Malohlava, and M. Schoeberl. Exhaustive testing of Safety-Critical Java. In *Proceedings of Java Technologies for Real-time and Embedded Systems*, pages 164–174. ACM, 2010.

[23] J. Kwon and A.J. Wellings. Memory management based on method invocation in RTSJ. In Robert Meersman, Zahir Tari, and Angelo Corsaro, editors, *OTM Workshops*, volume 3292 of *Lecture Notes in Computer Science*, pages 333–345. Springer, 2004.

[24] J. Kwon, A.J. Wellings, and S. King. Ravenscar-Java: a high-integrity profile for real-time Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):681–713, April 2005.

[25] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[26] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML. *ACM SIGSOFT Software Engineering Notes*, 31(3), 2006.

[27] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Compiler Construction*, pages 197–212. Springer, 2008.

[28] C. Marriott and A. Cavalcanti. SCJ: Memory-safety checking without annotations. In *FM 2014: Formal Methods*, pages 465–480. Springer, 2014.

[29] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.

[30] F. Nemer, H. Cassé, P. Sainrat, J.P. Bahsoun, and M. De Michiel. PapaBench: a free real-time benchmark. *WCET*, 4, 2006.

[31] K. Nilsen. A type system to assure scope safety within Safety-Critical Java modules. In *Proceedings of Java Technologies for Real-time and Embedded Systems*, pages 97–106. ACM, 2006.

[32] P. Parizek, T. Kalibera, and J. Vitek. Model checking real-time Java. Technical report, Citeseer, 2010.

[33] F. Pizlo and J. Vitek. Memory management for real-time Java: State of the art. In *11th ISORC*, pages 248–254, Orlando, FL, 2008. IEEE Press.

[34] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[35] J.R. Rios and M. Schoeberl. Hardware support for safety-critical Java scope checks. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on*, pages 31–38. IEEE, 2012.

[36] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[37] RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification, 1992.

[38] M. Saaltink. The Z/EVES System. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72 – 85. Springer-Verlag, 1997.

[39] M. Schoeberl. JOP: A Java optimized processor. *On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops*, pages 346–359, 2003.

[40] M. Schoeberl. Memory management for Safety-Critical Java. In Andy J. Wellings and Anders P. Ravn, editors, *Java Technologies for Real-time and Embedded Systems*, pages 47–53. ACM, 2011.

[41] M. Schoeberl and J.R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 54–61. ACM, 2012.

[42] A. Sherif, A. Cavalcanti, H. Jifeng, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, 2010.

[43] F. Siebert. Proving the absence of RTSJ related runtime errors through data flow analysis. In *Proceedings of the 4th international workshop on Java Technologies for Real-time and Embedded Systems*, pages 152–161. ACM, 2006.

[44] N.K. Singh, A.J. Wellings, and A. Cavalcanti. The cardiac pacemaker case study and its implementation in Safety-Critical Java and Ravenscar Ada. In *Proceedings of the 10th international workshop on Java Technologies for Real-time and Embedded Systems*, pages 62–71. ACM, 2012.

[45] D. Tang, A. Plsek, and J. Vitek. Static checking of Safety-Critical Java annotations. In *Proceedings of Java Technologies for Real-time and Embedded Systems*, pages 148–154. ACM, 2010.

[46] The Open Group. SCJ technology specification (v0.94). Technical report, June 2013.

[47] U.S. Department of Transportation. *Lane Departure Warning Systems (LDWS)*. `http://www.fmcsa.dot.gov/facts-research/research-technology/report/\lane-departure-warning-systems.htm`.

[48] Volvo. *The all-new Volvo V60 sports wagon blends style and performance with ground-breaking safety*. `http://www.volvocars.com/uk/top/about/news-events/Pages/default.aspx?itemid=50`.

[49] A.J. Wellings. *Concurrent and real-time programming in Java*. Wiley, 2004.

[50] A.J. Wellings, M. Luckcuck, and A. Cavalcanti. Safety-Critical Java level 2: motivations, example applications and issues. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 48–57. ACM, 2013.

[51] J. Woodcock and A. Cavalcanti. The semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, pages 184–203. Springer, 2002.

[52] J. C. P. Woodcock. *Using Z – Specification, Refinement and Proof.* Prentice-Hall, 1994. To appear.

[53] F. Zeyda, A. Cavalcanti, and A.J. Wellings. The Safety-Critical Java mission model: A formal account. In Shengchao Qin and Zongyan Qiu, editors, *ICFEM*, volume 6991 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2011.

[54] F. Zeyda, L. Lalkhumsanga, A. Cavalcanti, and A.J. Wellings. Circus models for Safety-Critical Java programs. *The Computer Journal*, 2013.