Model Checking of State-Rich Formalisms

By Linking to Combination of State-based Formalism and Process Algebra

Kangfeng Ye

Doctor of Philosophy

University of York

Computer Science

August 2016

Abstract

Computer-based systems are becoming more and more complex. It is really a grand challenge to assure the dependability of these systems with the growing complexity, especially for high integrity and safety critical systems that require extremely high dependability. *Circus*, as a formal language, is designed to tackle this problem by providing precision preservation and correctness assurance. It is a combination of Z, CSP, refinement calculus and Dijkstra's guarded commands. A main objective of *Circus* is to provide calculational style refinement that differentiates itself from other integrated formal methods.

Looseness, which is introduced from constants and uninitialised state space in *Circus*, and nondeterminism, which is introduced from disjunctive operations and CSP operators, make model checking of *Circus* more difficult than that of sole CSP or Z. Current approaches have a number of disadvantages like nondeterminism and divergence information loss, abstraction deterioration, and no appropriate tools to support automation. In this thesis, we present a new approach to model-check state-rich formalisms by linking them to a combination of a state-based formalism and a process algebra. Specifically, the approach illustrated in this thesis is to model-check *Circus* by linking to $CSP \parallel B$. Eventually, we can use ProB, a model checker for B, Event-B, and $CSP \parallel B$ etc., to check the resultant $CSP \parallel B$ model.

A formal link from *Circus* to $CSP \parallel B$ is defined in our work. Our link solution is to rewrite *Circus* models first to make all interactions between the state part and the behavioural part of *Circus* only through schema expressions, then translate the state part and the behavioural part to B and CSP respectively. In addition, since the semantics of *Circus* is based on Hoare and He's Unifying Theories of Programming (UTP), in order to prove the soundness of our link, we also give UTP semantics to $CSP \parallel B$. Finally, because both ends of the link have their semantics defined in UTP, they are comparable.

Furthermore, in order to support an automatic translation process, a translator is developed. It has supported almost all constructs defined in the link though with some limitations.

Finally, three case studies are illustrated to show the usability of our model checking solution as well as limitations. The bounded reactive buffer is a typical *Circus* example. By our model checking approach, basic properties like deadlock freedom and divergence freedom for both the specification and the implementation with a small buffer size have been verified. In addition, the implementation has been verified to be a refinement of the specification in terms of traces and failures. Afterwards, in the Electronic Shelf Edge Label (ESEL) case study, we demonstrate how to use *Circus* to model different development stages of systems from the specification to two more specific systems. We have verified basic properties and sequential refinements of three models as well as three application related properties. Similarly, only the systems with a limited number of ESELs are verified. Finally, we present the steam boiler case study. It is a real and industrial control system problem. Though our solution cannot model check the steam boiler model completely due to its large state space, our solution still proves its benefits. Through our model checking approach, we have found a substantial number of errors from the original *Circus* solution. Then with counterexamples during animation and model checking, we have corrected all these found errors.

Contents

List of Tables

List of Figures

Accompanying Materials

This thesis is accompanied by a CD, which contains an electronic version of this thesis and a number of additional materials listed in Appendices.

▷ "Model Checking of State-Rich Formalisms By Linking to Combination of State-based Formalism and Process Algebra": electronic version of the thesis in the PDF fromat

▷ "D.3 - CSP Libraries": corresponding CSP libraries as stated in Appendix D.3

▷ "F Translator": Java source code (.java format) and a latest release V0.5 (.jar format) of the translator *Circus2ZCSP* as stated in Appendix F

▷ "G Reactive Buffer": *Circus* models and resultant $CSP \parallel_B Z$ models for both specification and implementation of reactive buffer case study as listed in Appendix G

▷ "H ESEL": *Circus* models and resultant $CSP \parallel_B Z$ models for the Specification, the System One, and the System Two of ESEL case study as listed in Appendix H

▷ "I Steam Boiler": *Circus* model and resultant $CSP \parallel_B Z$ model for steam boiler case study as listed in Appendix I

Alternatively, these materials are also available online at

↬ Circus Programs at Github (https://github.com/RandallYe/Circus-Programs)

↬ Circus2ZCSP at Github (https://github.com/RandallYe/Circus2ZCSP)

though online site might have newer versions of all materials.

Acknowledgements

Making transition between academia and industry is not always easy. It is more difficult especially for me to return from industry to university to start a new journey in computer science, particularly pursuing a PhD in the area of formal methods. Without help from people I met in UK and support from my family, it is impossible for me to move forward in my research. Therefore, I would like to extend my sincere thanks to all persons who ever helped me either directly or indirectly, and either through communication or just knowledge I learn from their work. I am also grateful for the financial support from the Department of Computer Science to cover the tuition fee difference between home and overseas students.

First and foremost I would like to express my sincere gratitude to my supervisor, Prof. Jim Woodcock. This thesis represents not only my work but also the close work with Jim together. Since the first time I met Jim in 2012, I have been so impressed by his knowledge in UTP, CSP and Z etc. and also his attitude to research. His commitment to supervision is extremely important and helpful to me. In approximately four years since October 2012, we had more than one hundred supervision meetings and nearly thirty meetings for each academic year. Considering he is the head of department at this time, I fully understand it is very hard for Jim to commit to supervision in this level. I really appreciate it. I am also grateful for his patience and instructions. Since I do not have much knowledge in Z, CSP and UTP before, in my first year of PhD Jim gave me a lot of instructions and answered many questions from me when I was reading these books chapters by chapters. In addition, he plays a very important role in the decision of my research direction and PhD research schedule. The review of my work including this thesis is another great help from Jim. It is my pleasure to be a student of Prof. Jim Woodcock.

My work is also based on dozens of work from *Circus* team members, either previous or current, who I wish to thank. Without solid theoretical and practical work from them, it is impossible for me to begin my work in this direction. Their great work inspired me to start exploring *Circus*, especially the work about *Circus* refinement calculus and denotational semantics mainly from Prof. Ana Cavalcanti and Dr. Marcel Oliveira, and the work about operational semantics, model checking and *Circus* extension in CZT mainly from Dr. Leo Freitas.

I wish to thank my assessor, Dr. Jeremy Jacob. As a member of my Thesis Advisory Panel, he really cares about my research plan and progress, and gives me a lot of advice about that. His review comments about *Circus*, LaTeX, English, and schedule are very helpful. Furthermore, part of my work has been reviewed by a number of anonymous reviewers who I also wish to acknowledge.

My work relies on the model checker, ProB, to check the resultant linked models. I am grateful to the ProB team who develops this great tool which gives me insight into what a model checker can be. Its capability to support a wide variety of languages and the way to animate are very impressive. In addition, my work has benefited from the translator from Z to B in ProB too. Particular thanks to Prof. Michael Leuschel and Ivaylo Dobrikov for help and discussion about $CSP \parallel B$ support in ProB.

In addition to research, I also enjoy part-time work in Rapita Systems very much for three years during my PhD. People in Rapita are very friendly to me and professional at work. Particularly help from the development team is really appreciated. Special thanks to Dr. Guillem Bernat, Dr. Antoine Colin, Dr. Ian Broster, Dr. Zoë Stephenson, Dr. Will Lunniss, Chris Bryan, and Kathy Hemingway for their help.

To my best friend Xueyi Zou, we have been in York and started new PhD journey at the same time. We also share an office in the department. One year later, I and my wife Miao also met Xueyi's girl friend, Yuqi Chen. Four of us spent a lot of time together to enjoy food, chat and travel. I and Miao witnessed both of you got married in UK. Good luck to Xueyi for his thesis and Viva, and wish both of you have a bright future.

My wife Miao is the most important person I would like to say “thank you”. She is a lovely and wonderful wife to me. Without her encouragement and support, I cannot image what my life could be. Her company during my PhD is the source of my passion for research. In order to support my study as well as our life in UK, she has to move to another city to take a full-time job. She always works very hard and is very professional at work. Her commitment to work is very impressive. Then however we have to be apart frequently. Sometimes I travelled to her city. Sometimes she came back to the university on Friday and returned to work on Sunday. This is the most difficult time for us. Due to this, we really cherish our time together. I truly appreciate my wife's full support and understanding. Moreover, her encouragement also plays a significant role in my research. In the early stage of my PhD, she cannot fully understand the reason why I choose formal methods as my research topic, instead of real time or embedded systems that I excels as a senior embedded software engineer before my study. After two years, she has been convinced by our continuous discussion of benefits that formal methods could bring to software engineering. Then she also explained my research to her colleagues and friends when they asked her about my research. I might believe she can explain better than me to them since her description looks very easy to be understood from others' aspect. During the most difficult time of my research journey, Miao's help and support are extremely beneficial to me. I have married a person who I cannot live without and the person I have in my life is perfect for me. I would like to thank my parents-in-law for their support as well.

In the end, thank you to my parents, grandparents and brothers for their unconditional love and support to me throughout my life. My memorable moments with them, after I return to China occasionally during my PhD study abroad, are always unforgettable.

All the best to all.

In memory of my granny, I miss you.

Declaration

I hereby delcare that this thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own original work, except where otherwise stated. Other sources are acknowledged by explicit references. Chapter 4 and the reactive buffer case study in Chapter 7, partially based on the previously published paper [1], are an extended and revised version.

Chapter 1

Introduction

This chapter gives a brief context of our work presented in this thesis. To begin with, the specific research problems and our motivations of this work are provided. Then we present objectives and contributions of this research. In the end, the document structure is outlined.

1.1 Formal Methods and *Circus*

Computer systems, or computer-based systems, are becoming more and more complex during evolution, not only for individual physical systems but also for systems of systems. Computer systems have evolved from early generations (vacuum tube, transistor, IC, microprocessor) to desktop computers with internet, systems with ubiquitous computing [2], and recently cyber physical systems [3]. The growth of complexity in systems is due to 1) powerful and complex computers, 2) increasing functionality, 3) growing integration and interoperability, and 4) dynamic system and increasing dependability requirements.

Formal methods is a mathematically based technique to specify, develop, and finally verify systems. Because of its underlying mathematical theory, it can reason about systems from specification to implementation without ambiguity, which makes it particularly suitable to address this growing complexity issue.

Traditional researches in formal methods often focus on two schools: state-based, model-oriented specification languages such as Z [4], B [5] and VDM [6]; and behaviour-oriented process algebras such as CSP [7, 8], CCS [9] and ACP [10]. However, due to the growing requirements of integration, heterogeneity and interoperability, these formal methods have their inability to model increasingly complex systems. Therefore, in recent decades, there is rising research interest in specification languages that integrate both state and behavioural aspects.

Early solutions aim to combine them together, such as CSP-OZ [11], ZCCS [12], CSP-Z [13, 14], and $CSP \parallel B$ [15]. Fischer [16] gave a summary of combination solutions of Z and process algebras. These approaches provide the capability to specify systems from both state and behavioural parts and possibly verify them as well. For CSP-OZ, a translation [17] has been proposed to transform CSP-OZ specification into CSP, and then model check by FDR [18], a model checker for CSP. To verify the specification in CSP-Z, a similar solution [19] is presented to model the Z part as a CSP process and finally transform the CSP-Z specification fully to the CSP specification. Eventually, the CSP specification is model checked by FDR too. Particularly, $CSP \parallel B$ does not use the same method for verification. It is able to be model checked by ProB [20], a model checker and animator originally developed for the B method.

Though they have specification and verification capabilities, one gap between them is that there is no straight development strategy. According to the description above, a formal language needs the capabilities to specify, develop, and verify systems, and at the same time consistency between each stage should be preserved. The development method, well known as refinement, is a process to refine abstract specification into concrete implementation. In each step, it tends to eliminate non-determinism and give more specific details. This is a very important feature for formal methods to be applied to real industry. It is also one of main objectives of the development of Concurrent Integrated Refinement CalculUS (or, *Circus* for short) [21].

1.1.1 *Circus*

The objectives of *Circus* are manifold. First of all, as a formalism, it has very abstract constructs, which makes it capable of specifying and modelling systems at the level close to users' perception. Secondly, it can be very specific and detailed too to record the development and design decisions, which enables it to model systems at the level close to the final code. Then in order to bridge the gap between these two ends, it is very necessary to support refinement. The refinement notation makes *Circus* very useful to support the step-wise development of systems from the high-level specification to the low-level implementation. Particularly, *Circus* defines refinement in a calculational style [22]. Actually the integrated formal methods' refinement deficiency has motivated the creation of *Circus* [23]. Furthermore, its capability to specify systems not only from the state aspect but also from the behavioural part enables it to model large scale complex systems. Last but not least, to address the heterogeneity challenge, *Circus* bases its semantics on Hoare and He's Unifying Theories of Programming (UTP) [24], a common framework for the unification of programs from different paradigms. And *Circus* has extended to support time [25], object-orientation [26], mobility, etc.

To achieve these objectives, *Circus* is naturally designed to be a state-rich formalism and to specify systems from both state and behavioural aspects. It is a combination of Z, CSP, refinement calculus [22] and Dijkstra's guarded commands [27]. Z notation is used to specify the data part of *Circus* specification at a high level of abstraction, while CSP is to specify reactive and concurrent behaviour. In order to model the implementation of systems, the syntax constructs introduced in *Circus* include guarded commands, assignment, and variable block, etc.

Since *Circus* is a combination of several different languages with different semantics, there arises an issue about how to unify its semantics into one. The solution of CSP-OZ [28] is to introduce the failures-divergences model [8] to Object-Z [29] classes and then integrate CSP processes with Object-Z classes based on the same failures-divergences semantics. The solution of $CSP \parallel B$ is to treat a B machine as a CSP process and give CSP traces, stable failures and failures-divergences semantics to B machines [15]. *Circus* needs to combine not only the state-based language Z with the process algebra CSP, but also the refinement calculus with CSP and Z. Thus the solution for *Circus* is to formalise its model in UTP.

1.2 Formal Verification

In the last two decades, formal verification, the process to check if systems specified in formal languages satisfy their requirements, has been widely used in both academia and industry, especially in safety critical systems. And it has been becoming increasingly important due to the growth of complexity in these systems. Theorem proving [30], a technique to prove mathematical theorems using logic deduction, and model checking [31], another technique to check whether the system model in formal languages satisfies a desired specification by exhaustively searching the system's state space automatically, are two commonly used techniques for formal verification.

One major advantage of theorem proving over model checking is its capability to prove the correctness of systems regardless of their state space size. However it is a major challenge for model checking, known as the *state explosion problem* [32, 33]. Despite this advantage, theorem proving has its disadvantages as well. The expertise requirement for users of theorem prover is very high, and learning curve to use these tools could be very steep. Additionally, the users must know any details of the systems in proving. These two factors make theorem proving hard to apply in the very early stage of development when system requirements might be not all defined and clear. Comparatively, model checking could be easily used to separate modules as well as final systems. Its automated checking procedure, potential animation, counterexamples for debugging, and possible temporal logic checking, can help users to gain better understanding of systems and finally improve system requirements even in the early stage of development. It also can find errors in the design and give clues at the same time through counterexamples. Furthermore, refinement checking and properties checking, including safety, liveness, and others described in temporal logic, are beneficial to systems in design by providing guarantees that one implementation is a correct refinement of the specification and these properties hold. Hence, it is very important for a formalism to support both model checking and theorem proving in a complementary way. Finally, systems that are specified by this formalism could benefit from model checking and theorem proving together. To some extent, it proves the correctness of designed systems because the systems have been checked before being made.

1.3 Problems and Motivations

When it comes to formal verification for *Circus*, a number of approaches or tools have been developed from both theorem proving and model checking aspects.

1.3.1 Theorem Proving

From the theorem proving perspective, Oliveira *et al.* [34–36] present the mechanisation of UTP theories in a theorem prover, ProofPower-Z [37], which forms the basis of the mechanisation of *Circus* and its refinement calculus into this theorem prover as well. Then Zeyda *et al.* [38, 39] present a semantic embedding of the UTP framework in ProofPower-Z, extends the existing mechanisation work by Oliveira, and encodes a *Fib* process [39] into ProofPower-Z.

Furthermore, Feliachi *et al.* [40] have developed a machine-checked, formal semantics based on a shallow embedding of *Circus* in Isabelle/Circus, and the semantic theory of the UTP is also based on Isabelle/HOL [41]. Additionally, Foster *et.al* [42, 43] has introduced Isabelle/UTP, a novel mechanisation of Hoare and He's UTP in Isabelle/HOL as well. Isabelle/UTP is a deep embedding of the semantic model of UTP's alphabetised predicates. It differentiates itself from Isabelle/Circus in three principle ways: a unified type of alphabetised predicates, highly flexible encoding of predicates, and meta-level reasoning.

1.3.2 Model Checking

Since *Circus* is a combination of Z, CSP, refinement calculus and Dijkstra's guarded commands, its model checking is intrinsically more complicated and difficult than that of individual Z and CSP. The complexity of model checking *Circus* is increased due to two main factors. The first one is state space explosion challenge. Basically, the state of a system specified by *Circus* is the state of its processes. However, for each process it may contain state and behaviour, and consequently its state is a combination of both variable state and action state. In addition, the process's variable and action states are dynamically constructed and destroyed along with invocation and termination of the process. It possibly has an infinite number of distinct states as well. Because of this hierarchical structure of *Circus* and possible infinite states, how to represent and search its state space including infinite state space and infinite data type efficiently is really a challenge. Another factor is *Circus*'s very rich notations from Z, CSP and guarded commands. Along with its powerful expressiveness and high-level abstraction, all make the development of its model checker—to parse and type check *Circus* programs, check deadlock and livelock, check refinement in terms of *Circus* action refinement and data refinement, and CSP failure-divergence refinement—difficult.

From the model checking perspective, a number of approaches are proposed. We describe them briefly in this section and will give a review in depth in Chapter 3.

The first solution presented by Freitas [44] in his PhD thesis is a refinement model checker based on automata theory [45] and operational semantics of *Circus*. The operational semantics and the underlying automata theory have been formalised in **Z/Eves** [46], a theorem prover. A model checker architecture is also presented and a prototype has been developed in Java. Another operational semantics based model checker [47, 48] is built on the Microsoft FORMULA framework [49].

Apart from operational semantics based solutions, as far as we know, there are three other related work in addition to our approach [1]: *JCircus* [35, 50, 51], the *link* from *Circus* to CSP [52, 53], and the *map* of *Circus* processes and refinement to CSP processes and refinement [54, 55]. *JCircus*, as well as its extension *JCircus* 2.0 [56], translates a concrete *Circus* program to a Java program with JCSP [57], an implementation of CSP in Java. And the *link* is based on *JCircus*. Instead of JCSP, it links *Circus* to CSP_M [58], a machine-readable CSP on a functional language. The key point of the *map* is to transform stateful *Circus* programs to stateless *Circus* programs by introducing the memory model [59], and then convert stateless *Circus* to CSP_M. Both the *link* and the *map* use the refinement checker FDR [60] to model-check the resultant CSP.

From a theoretical point of view, the refinement model checker proposed by Freitas obviously is the ideal solution. It is originally designed for the *Circus* language, and all constructs are expected to be supported. Its architectural capability to integrate with a theorem prover is also a major advantage. However, the critical problems are the issues about the difficulties of developing a model checker for *Circus* as described above. How to run the model checking algorithms in parallel to achieve high performance and efficiency is a real challenge. Consequently, the prototype has not got further development. Other solutions as well as our approach actually are a compromise between idealism and realism. The model checker on FORMULA intends to check basic properties as well as temporal logic formulas, but it is not capable of refinement checking and animation. *JCircus* actually is not a model checking solution but an implementation instead, and it is restricted to executable *Circus* programs because Java is an imperative programming language and not a high-level specification language. Therefore, before supplying *Circus* programs to *JCircus*, it has to be refined to concrete programs. The *link* and the *map* are similar to transform both state and behavioural parts of *Circus* to CSP, which means all states are maintained in CSP. It is restricted to divergence-free *Circus* and the abstraction of *Circus* is sacrificed because the data part in *Circus* is modeled by Z which supports abstract data type but in CSP it is concrete. Furthermore, it is not convenient and capable in CSP to maintain very complex states, and rather difficult to understand the final CSP specification if it contains a lot of state operations. Finally, a lack of practicable translators for the *link* and the *map* is another obstacle. Though a prototype translator [53] has been developed for the *link*, only a small subset of *Circus* constructs is supported.

1.3.3 Our Approach

Circus is a formal language naturally suitable for the formal development of state-rich reactive systems because of its intrinsical basis on the rich notation Z, CSP, refinement in a calculational style, and guarded commands. In these reactive systems, the state is rich and actually plays a very important role across the development. As a result, one of our expectation to the model checker is to maintain the consistency of the state representation and refinement along the refinement from the specification. And our vision is to model-check all levels of *Circus* developments, from abstract specification, to each refinement, and to the final concrete implementation in a practicable way. Though current solutions of model checking *Circus* have some advantages, they do not meet our expectation very well.

Our approach as presented in the paper [1], as well as in this document, is motivated by current problems and expectations. Our solution is to link a *Circus* program, no matter what it is — a specification, refinement or implementation, to the combination of a state-based formalism and a process algebra. By this way, the state part of *Circus* is still modelled in a formal language which is capable of specifying systems from both abstract and concrete aspects. To be more specific, the integrated formal language we choose is the combination of B and CSP, $CSP \parallel B$. Finally, the resultant $CSP \parallel B$ program is model-checked by ProB.

The main difference of our approach from other solutions above is the representation of the state part of *Circus*. We use the B method, instead of Java in *JCircus* and CSP in the *link* and the *map*, to maintain the high-level abstraction in *Circus*.

Though our solution defined in this document is highly specific for the link between state-rich *Circus* and $CSP \parallel B$, its underlying principles are also applicable for other state-rich formalisms, such as Compass Model Language (or, CML for short) [61]. And the target language can be other combinations of state-based formalism and process algebra and not restricted to $CSP \parallel B$.

1.4 Objectives and Contributions of this work

According to our expectations and motivations of this work, our final goal is to have a model checking solution which is able to verify the *Circus* programs in different stages of development. For each individual program, such as a specification, a refinement, or an implementation, the model checker can verify some properties: deadlock free, livelock free, and other properties in the temporal logic, etc. Then for the development, it is capable of verifying the correctness of refinements. Obviously, there are always problems or errors in design. Thus an explicit and efficient way to give clues, for example counter examples, to the problems found during model checking is extremely helpful. Additionally, the capability to animate a *Circus* program is also very important, especially in the early stage of development. Both debugging and animation are beneficial to better understanding of the systems and, in turn, improvement of the design.

Our first contribution is to define a formal link from *Circus* to $CSP \parallel B$ and the soundness of the link is based on the UTP semantics. As the denotational semantics of *Circus* is based on UTP, we give the UTP semantics to $CSP \parallel B$ as well. As far as we know, this is original. This link is decomposed into several stages or functions, namely the rewrite of the *Circus* program to separate the state part from the behavioural part, the transformation of the state part to B, and the transformation of the behavioural part to CSP. To the best of our knowledge, separation of state and behaviour from *Circus* by rewriting, and the quite complete conversion of *Circus* expressions, predicates, and operators (even schema and free type) to CSP$_M$ are the first. The rewrite or transformation of *Circus* in the first stage can be useful for the development of other tools in the future to facilitate syntactic and semantic transformations to reduce substantial constructs. Some of rewrite rules could be used directly, and others might give insights into the considerations when transforming them. And the conversion of *Circus* expressions, predicates, and operators in Z and implementation of them in CSP$_M$ as additional libraries could be helpful even for the other solutions (the *link* and the *map*) because they face the similar challenge to implement all counterparts in CSP$_M$ as well.

The second contribution of this work is a translator, developed to link *Circus* programs to $CSP \parallel B$, which supports nearly all constructs that are defined in our link. The translator is written in Java and based on Community Z Tools (or, CZT for short) [62], a open framework to support a number of Z extensions. Our implementation decisions made for a number of *Circus* constructs and their transformations, and lessons learned during development could be very helpful for other *Circus* tool developers in the future. Especially, the solutions and decisions listed below could be very interesting to other developers.

Inheriting sections a solution is used to resolve and include all parent sections.

Schema as predicate a method is designed to convert schemas as predicate in *Circus* to equivalent predicates without normalisation [63] (normalisation is not expected when converted into CSP because it results in very basic types, such as $\mathbb{Z}$, which are difficult to be modelled in CSP_M and checked on ProB).

Schema declaration list a scheme is developed to expand schemas to get a list of variables declared along with their type without normalisation.

Global definitions reduction an algorithm is applied to resolve global and local schema definitions—global definitions are duplicated into each basic process in which they are referred, and no reference of global definitions are removed.

Construct references resolving an algorithm is delicately designed to maintain a reference map of all definitions and schemas in *Circus* to determine which should be converted to CSP_M and which should not.

Genericity in a nongeneric way there is a method to transform all generic constructs into nongeneric counterparts.

Furthermore, our translator *Circus2ZCSP* is bigger than other translators, such as Z2B and Z2Alloy, in CZT project. Therefore, it might be helpful for other CZT developers.

The third contribution of this work is three case studies which are formalised or partially formalised, which demonstrates the usability of our approach. The distributed reactive buffer case presented in the paper [64] is developed using the refinement strategy of *Circus* and regarded as verified. Our work has checked it again using model checking and proved its deadlock free and livelock free in terms of failures mode, and the refinement between the original specification and the final distributed buffer with specific constants configuration. The Electronic Shelf Edge Label (or, ESEL for short) is another case we present originally. The specification is illustrated in Section 2.3 of the background chapter. This aims to give audiences a straight sense of what a *Circus* specification looks like, just following the *Circus* syntax introduction. This case shows how a *Circus* program can be developed and verified using our approach in different levels of refinement. The last case is the *Circus* solution to the steam boiler control system [65]. The original *Circus* is proposed by Woodcock and Cavalcanti in the report [66] and the paper [67]. Our work has corrected and improved the steam boiler solution in *Circus*, and finally it could be regarded as a benchmark of *Circus*.

The last but not the least is the lessons we learned and experience we got during the development of this work. This gives insights into the design and development of similar tools for *Circus* or other state-rich formalisms.

1.5 Thesis Structure

Our work is to link *Circus* to $CSP \parallel B$. Therefore, a basic knowledge of *Circus*'s underlying semantics—UTP, and $CSP \parallel B$, is necessarily given in Chapter 2. Then in Chapter 3, we review current model checking solutions, explain our approach in depth, and give the considerations of decision made for our approach.

After that, our formal link of *Circus* to $CSP \parallel B$ is defined in Chapter 4. In the beginning of this chapter, the link strategies and decomposition are presented. Then individual rules for each decomposed function are defined. It is worth nothing that these rules are individually based and the composed relation of the link is given in Appendix E.

The soundness of our link is explained in Chapter 5. Our principle is to give the UTP semantics to both *Circus* constructs and the linked $CSP \parallel B$ constructs, then a comparison of their semantics is taken to determine if the rule is sound or not.

Chapter 6 presents a translator we develop for this work. Its architecture and a number of algorithms are described in details.

In Chapter 7, we describe three case studies of the application of our approach in ascending complexity order. The first case, a bounded buffer, is the simplest one and has been developed as a typical example for *Circus*. And the second one, a ESEL system, is our original work. Then the third one, the steam boiler control system, is an industrial classic example and a benchmark for formal languages. We also illustrate model checking experiment results of these cases in the chapter.

Finally, we conclude this document in Chapter 8 and give perspectives.

Apart from all these chapters in the body of the document, there is also an important appendices part. In Appendix A, the *Circus* syntax is illustrated. Appendix B lists all definitions in the document. In Appendix C, some auxiliary theorems, laws, and lemmas are illustrated along with their proofs. Then the maps of all expressions, predicates, and operators in *Circus* to CSP_M, as well as their implementation in CSP libraries, is listed in Appendix D. Appendix E composes the individual rules defined in Chapter 4 and gives direct links from *Circus* constructs to the combination of CSP and Z, which are used to reason about the soundness of the link in Chapter 5. After that, we present the *Circus* models of three case studies in Appendix G, Appendix H, and Appendix I.

Chapter 2

Background

This chapter presents the background of our work. Since *Circus* is a combination of Z, CSP, refinement calculus and guarded commands, in order to combine these languages together, *Circus* bases its semantic model on the Unifying Theories of Programming (UTP). Therefore, Section 2.1 first briefly introduces UTP including the alphabetised relational calculus, and the theories of designs, reactive processes, and CSP processes. And after that in Section 2.2, we give an introduction of *Circus* in detail from its syntax to semantics, then present the *Circus* specification of an ESEL system as an example in Section 2.3 to illustrate the basic structure of a *Circus* model. Finally, the basic knowledge of $CSP \parallel B$ is presented in Section 2.4.

2.1 Unifying Theories of Programming

UTP is a unified framework and theoretical basis for describing and specifying computer languages across different paradigms such as imperative, functional, declarative, nondeterministic, concurrent, reactive and high-order. A theory in UTP is described from three parts: *alphabet*, *signature* and *healthiness conditions*. *alphabet* denotes a set of variable names that gives the vocabulary of the theory to be studied; *signature* gives the rules of primitive statements of the theory and how to combine them together to get more complex programs; and *healthiness conditions* are a set of mathematically provable laws or equations to characterise the theory.

2.1.1 Alphabetised Relational Calculus

The alphabetised relational calculus [68] is the most basic theory in UTP. The denotational semantics of programs are given as relations between an initial observation and a subsequent observation, which might be either an intermediate observation or a final observation. A relation is defined as a predicate P with undecorated variables (v) and decorated variables (v') as its alphabet, abbreviated as αP. v denotes the observation made initially and v' denotes the observation made at the intermediate or final state. αP consists of an input alphabet $in\alpha P$ (v) and an output alphabet $out\alpha P$ (v').

Standard predicate operators can be used to combine alphabetised predicates together to construct more complicated programs. In UTP, a condition is a predicate without an output alpha. The infix conditional operator is defined as propositional logic operators

$$P \lhd b \rhd Q \mathrel{\widehat=} (b \land P) \lor (\neg b \land Q)$$

Sequence is defined as relational composition

$$P\left(v'\right) ; Q\left(v\right) = \exists\, v_0 \bullet P\left(v_0\right) \land Q\left(v_0\right)$$

provided $out\alpha P = in\alpha Q' = \{v'\}$. For sequence $P \,;\, Q$, its intermediate state cannot be directly observed. Therefore, the definition of composition above uses existential quantification to hide the intermediate observation.

Assignment upon an alphabet A, where $A = \{x, y, z\}$, is defined as a conjunction of equalities which states only the value of the variable x is changed.

$$x{:=}_A e \mathrel{\widehat=} \big(x' = e\big) \wedge \big(y' = y\big) \wedge \big(z' = z\big)$$

Nondeterminism is defined as choice, a disjunction

$$P \sqcap Q \mathrel{\widehat=} P \vee Q$$

where $\alpha P = \alpha Q$.

Variable block below introduces x and x' into Q

$$\textbf{\textit{var}}\ x \,;\, Q \,;\, \textbf{\textit{end}}\ x \mathrel{\widehat=} \big(\exists\, x, x' \bullet Q\big)$$

Recursion ($X = F(X)$) is defined as the weakest fixed point F, $\mu\, F$.

$$\mu\, F \mathrel{\widehat=} \bigsqcap\{X \mid F(X) \sqsubseteq X\}$$

where $\sqsubseteq$ is defined below.

2.1.2 Refinement Calculus

Furthermore, refinement calculus is another important concept in UTP concerned with program development to achieve program correctness. In UTP, the notation for program correctness is the same for every paradigm: in each state, the implementation P implies its specification S, which is denoted by $S \sqsubseteq P$. And $\sqsubseteq$ is defined as the universal closure of $P \Rightarrow S$ as follows. The universal closure of P, $[P]$ is an abbreviation for $(\forall\, x, x', y, y' \bullet P)$ provided the alphabet of P is equal to $\{x, x', y, y'\}$.

$$S \sqsubseteq P \quad \text{iff} \quad [P \Rightarrow S]$$

This definition means that the observation of the program P must be a subset of the observation permitted by the specification S. A refinement sequence is shown as follows.

$$\textbf{true} \sqsubseteq S1 \sqsubseteq S2 \sqsubseteq P1 \sqsubseteq P2 \sqsubseteq \textbf{false}$$

$S1$ is more general and abstract specification than $S2$ and thus more easier to implement. The predicate **true**, abortion, is the easiest one and can be implemented by anything. $P2$ is more specific and determinate program than $P1$ and thus $P2$ is more useful in general. **false**, miracle, is the strongest predicate and it is impossible to implement in practice.

An important law about refinement is that if a specification S can be refined by a program either P or Q, then it can be refined by the choice of them.

$$S \sqsubseteq P \vee Q \quad \text{iff} \quad S \sqsubseteq P \text{ and } S \sqsubseteq Q$$

2.1.3 Designs

Designs in UTP are a subclass of relations that deals with the non-termination problem of programs, which states that the composition of P with a **true** (a non-terminated program, $X = X$) in either order shall result in a non-terminated program, except if the program P is a **false** (a miraculous predicate).

$$\textbf{true} \,;\, P = \textbf{true} = P \,;\, \textbf{true}$$
$$\textbf{true} \,;\, \textbf{false} = \textbf{false} = \textbf{false} \,;\, \textbf{true}$$

In a design, two additional boolean variables are introduced into the alphabet of programs: $okay$, the observation that the program has been started, and $okay'$, the observation that the program has terminated. If $okay$ is false, then the program has not started yet and the initial state is not observable. If $okay'$ is false, the program has not terminated yet and the final state is not observable.

A design (written as $P \vdash Q$ [24, 69]) is defined as follows.

$$P \vdash Q \mathrel{\widehat{=}} \big(okay \wedge P \Rightarrow okay' \wedge Q\big)$$

If a program starts in a state where P holds, then it will terminate, and on termination Q is established.

For designs, there are four healthiness conditions listed in Table 2.1 where $P[false/okay']$ is the result of substituting $false$ for the variable $okay'$ in P.

2.1.4 Reactive Processes

The behaviour of a design is observed at the initial state and the final state of a program by relating its precondition to its postcondition. But a reactive process cannot be characterised from the final observation alone, because it interacts with its environment (other programs and users). It must take intermediate states into account. Therefore, three extra variables: tr, ref, and $wait$, and their dashed counterparts, in addition to $okay$ and $okay'$, are introduced. tr and tr' are finite sequences and record the traces. ref and ref' records a set of events that might be refused. Boolean variables $wait$ and $wait'$ records whether a process has terminated or in an intermediate state awaiting further interaction with the environment. A number of interesting combinations of these variables are listed below.

- $okay \wedge \neg wait$ - started in a stable state
- $okay \wedge wait$ - not started but in a stable state
- $\neg okay$ - not started but in an unstable state
- $okay' \wedge \neg wait'$ - terminated
- $okay' \wedge wait'$ - in an intermediate state
- $\neg okay'$ - in an unstable state

Three healthiness conditions of reactive processes, ***R1***, ***R2***, and ***R3***, are displayed in Table 2.1. They can be composed into ***R***.

$$\boldsymbol{R} \mathrel{\widehat{=}} \boldsymbol{R1} \circ \boldsymbol{R2} \circ \boldsymbol{R3}$$

A reactive process is defined as a relation that has $okay$, tr, ref, and $wait$, and their dashed counterparts in its alphabet, and satisfies the three healthiness conditions.

2.1.5 CSP Processes

CSP processes are defined as reactive processes with additional two healthiness conditions: ***CSP1*** and ***CSP2*** in Table 2.1. According to Definition 8.2.1 [24], a reactive process is a CSP process if it satisfies ***CSP1*** and ***CSP2***. However, to characterise a subset of UTP relations that are processes written in CSP notations, those two healthiness conditions are not strong enough. Three extra healthiness conditions: ***CSP3***, ***CSP4***, and ***CSP5***, are introduced in CSP.

The relationship between relations, designs, reactive processes and CSP processes are given in the tutorial [68], "CSP processes are reactive; moreover they are the ***R***-image of designs" [68, Figure 1, p.257]. An important theorem [68, Theorem 4.2] defines CSP processes as reactive designs.

Table 2.1: Healthiness Conditions

Name	Definition	Description
Designs		
H1	$P = (okay \Rightarrow P)$	Unpredictability. No prediction about the initial or final state of P before it has started
H2	$\left[\begin{array}{l} P\,[false/okay'] \\ \quad \Rightarrow P\,[true/okay'] \end{array}\right]$	Termination always possible. If a program P allows non-termination under certain conditions, then it also allows termination under the same conditions
H3	$P = P\,; II_D$	Dischargeable assumption. A design is ***H3*** if its precondition is a condition
H4	$P\,; \mathbf{true} = \mathbf{true}$	Feasibility. A design is ***H4*** if it is feasible to establish a final state
Reactive Processes		
R1	$P = P \wedge (tr \leq tr')$	Traces can be extended but never undo
R2	$P\,(tr, tr') = P\,(\langle\rangle, tr' - tr)$	The behaviour is unrelated to the past
R3	$P = (II_{rea} \lhd wait \rhd P)$	A process starts only after its predecessor has terminated
CSP Processes		
CSP1	$P = \begin{array}{l} P \vee \\ (\neg okay \wedge tr \leq tr') \end{array}$	In case of divergence, the only guaranteed property is extension of the trace
CSP2	$P = P\,; \left(\begin{array}{l} (okay \Rightarrow okay') \wedge \\ (v' = v) \wedge \\ (tr' = tr) \wedge \\ (ref' = ref) \wedge \\ (wait' = wait) \end{array}\right)$	A process may not require non-termination
CSP3	$P = SKIP\,; P$	The behaviour of P does not depend on the initial value of ref
CSP4	$P = P\,; SKIP$	In case of termination or divergence, the final value of ref' is irrelevant
CSP5	$P = P \mathbin{\vert\vert\vert} SKIP$	A deadlocked process which refuses some set of events offered by its environment will be still deadlocked in an environment which offers even fewer events
Circus Actions		
C1	$A = A\,; \mathbf{Skip}$	On termination or divergence, the value of ref' has no relevance
C2	$A = A \llbracket v \mid \varnothing \rrbracket \mathbf{Skip}$	A deadlocked action which refuses some set of events offered by its environment will be still deadlocked in an environment which offers even fewer events
C3	$A = \boldsymbol{R}\left(\neg A^f_f\,; true \vdash A^t_f\right)$	No dashed variables in the precondition of the reactive design form of a *Circus* action

Theorem 2.1.1. For every CSP process P, $P = \boldsymbol{R}\left(\neg P^f_f \vdash P^t_f\right)$.

In the theorem, P^t_f is an abbreviation of $P[\mathit{false}, \mathit{true}/\mathit{wait}, \mathit{okay}']$ that denotes P has started but does not diverge, and P^f_f is an abbreviation of $P[\mathit{false}, \mathit{false}/\mathit{wait}, \mathit{okay}']$ that denotes P has started but diverges. This theorem states that every CSP process can be defined in terms of a reactive design through $\boldsymbol{R}$.

2.1.6 Various Semantics

UTP is capable of presenting the mathematical theory from various styles, including denotational semantics, operational semantics and algebraic semantics [24]. The denotational theory characterises the program language from its observable aspects through mathematical definitions; the operational semantics of a programming language just takes into account a set of individual steps in execution, other than the overall observable effects of the program like denotational semantics; while the algebraic semantics provides a collection of algebraic laws in terms of equations and inequations which are used for comparison and transformation of designs, and optimisation of programs.

2.2 *Circus*

This section gives an introduction to *Circus* from its syntax to its semantics and refinement calculus. Finally, a specification example is illustrated.

2.2.1 *Circus* Syntax

The complete syntax of *Circus* is composed of two parts. The first part is the syntax of Z in the ISO Standard dialect(ISO Standard Z [70] or ISO Z for short). The concrete syntax of ISO Standard Z can be found in [70, clause 8]. It is omitted in this document for brevity. And the second part is the syntax introduced for the combined behaviour (CSP and guarded commands), and for the overall structure of a *Circus* specification. Its complete BNF syntax is illustrated in Figure A.1 in Appendix A, where $\mathbf{N}^*$ and $\mathbf{N}^+$ stand for the repetition of $\mathbf{N}$ zero or more times, and one or more times respectively. They are different from the syntactic metalanguage used in ISO Standard Z. In ISO Standard Z, it uses { **N** } to represent the repeat of **N** zero or more times. In the syntax, the syntactic category **N** is a valid Z identifier defined in the standard, while categories **Par**, **SchemaExp**, **Exp**, **Pred** and **Decl** represent Z paragraphs, schema expressions, expressions, predicates and declarations defined in the standard too. The syntax of *Circus* is a free mixture of CSP and Z. Particularly, for the behavioural part in CSP, its syntax is different from CSP$_M$. Unlike CSP$_M$ which is defined upon a functional language, CSP in *Circus* is defined using Z syntax as well.

Generally, a *Circus* specification consists of a sequence of paragraphs: Z paragraphs, channel declarations, channel set declarations or process declarations.

$$\begin{array}{ll} \text{Program} & ::= \text{CircusPar}^* \\ \text{CircusPar} & ::= \text{Par} \mid \textbf{channel}\ \text{CDecl} \mid \textbf{chanset}\ \text{N} == \text{CSExp} \mid \text{ProcDecl} \end{array}$$

2.2.1.1 Channel Declarations

All channels used in a *Circus* specification must be declared. Channels in *Circus* can be declared in several ways.

$$\begin{array}{ll} \text{CDecl} & ::= \text{SimpleCDecl} \mid \text{SimpleCDecl;CDecl} \\ \text{SimpleCDecl} & ::= \text{N}^+ \mid \text{N}^+ : \text{Exp} \mid [\text{N}^+]\text{N}^+ : \text{Exp} \mid \text{SchemaExp} \end{array}$$

To begin with, channels can be declared in the same way as that in CSP. A synchronisation channel declaration only gives the name of events and no value is communicated on the channels. While a typed channel declaration specifies not only the name of events but also the type of values on the channels. And all events in this declaration have the same type.

Additionally, a family of channels can be introduced by the generic channel declaration, in which specified types are provided as parameters to the channels. For instance, $\textbf{channel}[T]c : T$ declares a family of channels c with the formal parameter T. For a channel c with a specific type S of its value, it is instantiated by $c[S]$. It is worth noting that multiple instantiations of generic channels with the same name c and type S actually refers to the same channel. It means for a parallel composition $P \llbracket \mid CS \mid \rrbracket Q$, if the instantiation $c[S]$ occurs in the P, Q, and CS, the parallel composition must synchronise on the $c[S]$ event.

Finally, apart from the declarations above that multiple channels with the same type can be declared in one declaration, channels with different types can be grouped in a schema **SchemaExp** without the predicate part.

2.2.1.2 Channel Set Declarations

A channel set declaration relates a channel set name to a channel set expression **CSExp** which basically is a set of declared channels.

$$\begin{aligned} \mathrm{CSExp} \quad &::= \{\!|\ |\!\} \mid \{\!| \mathrm{N}^{+} |\!\} \mid \mathrm{N} \mid \mathrm{CSExp} \cup \mathrm{CSExp} \mid \mathrm{CSExp} \cap \mathrm{CSExp} \\ &\mid \quad \mathrm{CSExp} \setminus \mathrm{CSExp} \end{aligned}$$

It can be the empty channel set $\{\!|\ |\!\}$, channel set extensions like $\{\!| c, d, \cdots |\!\}$, references to other **CSExp**, or expressions defined upon other Z set operators, such as $CSExp \cup CSExp$, $CSExp \cap CSExp$, and $CSExp \setminus CSExp$.

2.2.1.3 Process Declarations

Processes can be declared in several different ways.

$$\begin{aligned} \mathrm{ProcDecl} \quad &::= \textbf{process}\ \mathrm{N} \mathrel{\widehat=} \mathrm{ProcDef} \mid \textbf{process}\ [\mathrm{N}^{+}]\mathrm{N} \mathrel{\widehat=} \mathrm{ProcDef} \\ \mathrm{ProcDef} \quad &::= \mathrm{Decl} \bullet \mathrm{ProcDef} \mid \mathrm{Decl} \odot \mathrm{ProcDef} \mid \mathrm{Proc} \end{aligned}$$

They can be parametrised processes, indexed processes, explicitly defined processes, compound processes defined in terms of CSP operators, or renamed processes by substituting new channels for old channels in the name list.

For a parametrised process ($PP \mathrel{\widehat=} x : T_x ; y : T_y ; \cdots \bullet P$), the parameters $(x, y, \ldots)$ are free in P. The parametrised process can be instantiated by supplying real expressions for their parameters like $P(e_1, e_2, \cdots)$. Particularly, the indexed process definition ($IP \mathrel{\widehat=} i : T \odot P$ where i does not occur in P and only one parameter i is allowed) and its instantiation ($IP\lfloor e \rfloor$) are new to *Circus*. The only difference between indexed processes and parametrised processes is that for each channel c (with the type T_c) in the indexed processes, it is renamed to $c\_i.e$, where the new channels c_i are implicitly declared in this definition like the declaration

$$\textbf{channel}\ c\_i : T \times T_c$$

The messages that are communicated on these implicit channels include two fields: the first one is the index element i with the type T, and the second one is the field from the original channel c with the type T_c. Furthermore, both parametrised processes and indexed processes can be defined anonymously and instantiated together:

$$(x : T_x ; y : T_y ; \cdots \bullet P)(e_1, e_2, \cdots)$$

and

$(i : T \odot P) \lfloor e \rfloor$

Explicitly defined processes, or basic processes, are the basic unit in a *Circus* specification.

$$\text{Proc} \quad ::= \textbf{begin}\ \text{PPar}^*\ \textbf{state}\ \text{N} \mathrel{\widehat=} \text{SchemaExp PPar}^* \bullet \text{Action}\ \textbf{end}$$

Basically, an explicitly defined process consists of a state paragraph, name set paragraphs, Z paragraphs, action definitions, and a main action. Among them, the main action is mandatory, and others are optional. They all together form the data part (the state paragraph and Z paragraphs) and the behavioural part (action definitions and the main action) of the process. All state components of this process are given in the state paragraph marked by **state**. And the predicate of the state schema puts extra constraint on state components like an invariant that must be always held within this process. The overall behaviour of this process is characterised by its main action. In addition, a name set paragraph within the process relates the name to a name set expression **NSExp** which means a set of variables. Similar to channel set expressions, name set expressions, used in parallel composition and interleaving of actions, can be the empty name set $\{\}$, set extensions like $\{s, l, \cdots\}$, references to other **NSExp**, or expressions defined upon other Z set operators, such as $NSExp \cup NSExp$, $NSExp \cap NSExp$, and $NSExp \setminus NSExp$.

$$\begin{array}{lll}
\text{Proc} & ::= & \cdots \\
& | & \text{Proc} \mathbin{;} \text{Proc} \mid \text{Proc} \mathbin{\Box} \text{Proc} \mid \text{Proc} \sqcap \text{Proc} \mid \text{Proc} \ \llbracket \text{CSExp} \rrbracket \ \text{Proc} \\
& | & \text{Proc} \mathbin{|||} \text{Proc} \mid \text{Proc} \setminus \text{CSExp} \mid \text{Proc}[\text{N}^+ := \text{N}^+] \mid \text{N}[\text{Exp}^+] \mid \text{N} \\
& | & (\text{Decl} \bullet \text{ProcDef}) \left(\text{Exp}^+\right) \mid \text{N} \left(\text{Exp}^+\right) \mid (\text{Decl} \odot \text{ProcDef}) \lfloor \text{Exp}^+ \rfloor \\
& | & \text{N} \lfloor \text{Exp}^+ \rfloor \mid \mathop{;} \text{Decl} \bullet \text{Proc} \mid \Box\, \text{Decl} \bullet \text{Proc} \mid \bigsqcap \text{Decl} \bullet \text{Proc} \\
& | & \llbracket \text{CSExp} \rrbracket\ \text{Decl} \bullet \text{Proc} \mid \big|\big|\big|\ \text{Decl} \bullet \text{Proc}
\end{array}$$

Processes can also be defined by combining declared processes in terms of CSP operators, such as sequential composition ;, external choice $\Box$, internal choice $\sqcap$, parallel composition $\llbracket\rrbracket$, interleaving $|||$, hiding $\setminus$, and relative finite iterated operators (;, $\Box$, $\sqcap$, $\llbracket\rrbracket$, and $\big|\big|\big|$). All these composition operators are the same as those in CSP. But it is worth noting that the parallel composition operator of processes follows the *generalised* or *interface* parallel operator [71, Section 4.3] in CSP. Therefore, when it comes to $P \llbracket\ cs\ \rrbracket\ Q$, for all events in cs, they must be synchronised between P and Q; and for all events outside of cs, they proceed independently.

A declared process can be referred in other processes by process invocation P which is just the name of the process.

Finally, processes can be declared to be generic as well like $\textbf{process}[T]GP \mathrel{\widehat=} P$ and instantiated like $GP[S]$ where S is a specific type.

2.2.1.4 Actions

In *Circus*, an action definition in an explicitly defined process links the name of the action to its body. As with processes, actions can be parametrised as well and afterwards they are instantiated. But unlike processes, actions cannot be defined as generics.

$$\begin{array}{lll}
\text{PPar} & ::= & \text{Par} \mid \text{N} \mathrel{\widehat=} \text{ParAction} \mid \textbf{nameset}\ \text{N} == \text{NSExp} \\
\text{ParAction} & ::= & \text{Action} \mid \text{Decl} \bullet \text{ParAction}
\end{array}$$

An action can be a schema expression as action $\left(SchExp\right)$, a command, an action invocation, a CSP action, or an action renaming.

$$\text{Action} \quad ::= \left(\text{SchemaExp}\right) \mid \text{Command} \mid \text{N} \mid \text{CSPAction} \mid \text{Action}[\text{N}^+ := \text{Exp}^+]$$

The variables in actions, including state variables and local variables, can be renamed through the action renaming. However, unlike the process renaming, channels in the action will not be changed.

An action invocation is a reference to the name of the action defined and actually it is equal to its body.

Schema Expressions as Action The schema expression as action is very important in *Circus*. It provides a way for the behavioural part of the process to interact with its state part. It behaves like an execution of the same name operational schema if the precondition of the schema holds and at the same time the postcondition of the operational schema is established. Or if its precondition does not hold, the action diverges. It is worth noting that the input and output variables of the schema must be in scope in the context of this action.

CSP Actions In *Circus*, actions can be defined in terms of CSP as well. **Skip**, **Stop** and **Chaos** are three primitive CSP actions for termination, deadlock and divergence.

$$\begin{array}{lll}
\text{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \text{Comm} \rightarrow \text{Action} \mid \text{Pred} \ \& \ \text{Action} \\
& \mid & \text{Action} \, ; \text{Action} \mid \text{Action} \, \Box \, \text{Action} \mid \text{Action} \sqcap \text{Action} \\
& \mid & \text{Action} \llbracket \text{NSExp} \mid \text{CSExp} \mid \text{NSExp} \rrbracket \text{Action} \\
& \mid & \text{Action} \, |\!\![\text{NSExp} \mid \text{NSExp}]\!\!| \, \text{Action} \\
& \mid & \text{Action} \setminus \text{CSExp} \mid \text{ParAction} \left(\text{Exp}^{+}\right) \mid \mu \ \text{N} \bullet \text{Action} \\
& \mid & ; \text{Decl} \bullet \text{Action} \mid \Box \, \text{Decl} \bullet \text{Action} \mid \bigsqcap \text{Decl} \bullet \text{Action} \\
& \mid & \llbracket \text{CSExp} \rrbracket \, \text{Decl} \bullet \llbracket \text{NSExp} \rrbracket \, \text{Action} \mid \,|||\, \text{Decl} \bullet |\!\![\text{NSExp}]\!\!| \, \text{Action} \\
\text{Comm} & ::= & \text{N CParameter}^{*} \mid \text{N} \ [\text{Exp}^{+}] \ \text{CParameter}^{*} \\
\text{CParameter} & ::= & ?\text{N} \mid ?\text{N} : \text{Pred} \mid !\text{Exp} \mid .\text{Exp}
\end{array}$$

A prefixing action $Comm \rightarrow A$ provides its environment with the communication $Comm$ at first, and after that, it behaves like the action A. A communication is an event that is a pair from the channel name to the value of messages. In *Circus*, the event name can be a defined channel or an instantiation of a defined generic channel like $c[T]$. And the message part of communication can be composed of several fields: input $?x$, constrained input $?x : P$ where P is a predicate, and output $.x$ or $!x$. And these fields can be combined together to form a complicated communication, such as $c?x!y?z : (z > 10).u$, to pass multiple objects at the same time. In particular, the constrained input $?x : P$ places extra constraint on the value that can be passed on the channel in additional to its type in the channel declaration. Therefore, the overall constraint on the field is like $\{x : T \mid P\}$. Nonetheless, there is different syntax in CSP for the constrained input. In CSP, its syntax is like $?x : S$ where S is a set expression. And the overall constraint is the intersection of its type T and the extra constraint S: $T \cap S$

An action can be guarded by a condition, a predicate without after-state variables. This construct is named guarded action, such as $(g) \ \& \ A$ where g is a guarded condition. If the condition is evaluated to true, it behaves exactly the same as A. Otherwise, if the condition does not hold, it deadlocks.

As with processes, actions can be composed in terms of CSP operators as well. For the sequence composition of two actions such as $A_1 \, ; A_2$, A_2 is hidden from its environment initially and available only after A_1 terminates. For internal choice and hiding of actions, they are similar to those in CSP. However, for external choice of actions, though it has the same syntax $A_1 \Box A_2$ as that in CSP, the resolution of the external choice in *Circus* is different. An external choice of actions are only resolved by events or termination. And a state change such as assignment and schema expression as action will not resolve the choice. For instance, the external choice

$$\left(\left(Sch\right) ; c \rightarrow \textbf{Skip}\right) \Box \left(x := 1\right)$$

can only be resolved by the events c, or the termination of the action $x := 1$. Recursions definition (μ) is also provided for actions though it is not the case for processes.

And especially parallel composition of actions ($A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$) and interleaving of actions ($A_1 \interleave[ns_1 \mid ns_2]\interleave A_2$) in *Circus* are slightly different from those in CSP. Generally, the parallel composition of actions follow the *generalised* or *interface* parallel operator [71, Section 4.3] in CSP as well. Both actions synchronise on the events in cs. But two name set expressions, ns_1 and ns_2, are introduced. They are disjoint state partitions. For the parallel composition, all variables in scope of A_1 and A_2 are accessible for both actions, but only changes made to the variables in ns_1 by A_1 and the variables in ns_2 by A_2 have final effect, that is, these variables might be possibly updated. Other variables not in both ns_1 and ns_2 remain unchanged. For interleaving, it is similar.

Similar to processes, iterated operators such as iterated sequential composition, iterated external choice, iterated internal choice, iterated parallel composition, and iterated interleaving are provided for actions as well.

Commands A command can be an assignment, an alternation, a guarded command, a variable block, a parametrisation by value, a parametrisation by result, a parametrisation by value-result, a specification state, an assumption, or a coercion.

$$\begin{array}{lll}
\text{Command} & ::= & \text{N}^{+} := \text{Exp}^{+} \mid \textbf{if}\ \text{GActions}\ \textbf{fi} \mid \textbf{var}\ \text{Decl} \bullet \text{Action} \\
 & \mid & \text{N}^{+} : [\text{Pred}, \text{Pred}] \mid \{\text{Pred}\} \mid [\text{Pred}] \\
 & \mid & \textbf{val}\ \text{Decl} \bullet \text{Action} \mid \textbf{res}\ \text{Decl} \bullet \text{Action} \mid \textbf{vres}\ \text{Decl} \bullet \text{Action} \\
\text{GActions} & ::= & \text{Pred} \rightarrow \text{Action} \mid \text{Pred} \rightarrow \text{Action} \,\Box\, \text{GActions}
\end{array}$$

An assignment action may assign one or more variables simultaneously. For instance, $x, y := 1, x$ will update the variable x to 1 and y to the before-state x at the same time, and the updated after-state x (equal to 1) is impossible to be seen in the assignment of y.

Alternation provides a way from branching in *Circus*. If none of their guarded conditions hold, the alternation diverges. If there is only one condition holds, then the alternation behaves like the action guarded by this condition. Or if more than one condition hold, the alternation is a nondeterministic choice of those actions whose conditions are true. For instance, the alternation, $\mathbf{if}\ g_1 \longrightarrow A_1 \mathbin{[]} g_2 \longrightarrow A_2\ \mathbf{fi}$, diverges if both g_1 and g_2 are not true; or it behaves like A_1 if only g_1 is true; or similarly, A_2 if only g_2 is true; or $A_1 \sqcap A_2$ if both g_1 and g_2 are true.

A variable block ($\mathbf{var}\, x : T \bullet A$) declares local variables, in its declaration part, that will be used in its action. Particularly, the initial value of x is arbitrarily chosen. And substitution by value, result and value-result from [22] are also introduced in *Circus* and they have the similar syntax with the variable block. But they are defined as parametrised actions and need invocations. The invocation of substitution by value, $(\mathbf{val}\, x : T \bullet A)\,(e)$, declares a local variable x like the variable block but the initial value of x is assigned to e where e contains no x. Similarly, the invocation of substitution by result, $(\mathbf{res}\, x : T \bullet A)\,(e)$, and the invocation of substitution by value-result also declares a local variable x like the variable block respectively. But for substitution by result, the initial value of x is arbitrarily chosen and the final value of x after A is assigned e where e must be a variable name. Especially substitution by value-result has the combined behaviour of substitution by value and substitution by result, and the initial value of x is assigned to e, and finally the final value of x is assigned to e.

Apart from substitutions, the specification statement [22] is introduced to *Circus* as well to write the *Circus* specification in an abstract way. A specification statement such as $w : [\, pre, post\,]$ means if the initial state of variables satisfies pre, then only variables in the frame w might be changed to form the final state of variables that satisfies $post$. Two additional simplified constructs, assumption $\{pre\}$ and coercion $[\, post\,]$, are simply abbreviations of $:[\, pre, true\,]$ and $:[\, true, post\,]$ separately.

2.2.2 *Circus* Semantics and Healthiness Conditions

The denotational semantics of *Circus* is presented in [21] at first, then improved in [72], and finalized in [73]. In addition, the semantics of *Circus* has been mechanised in ProofPower-Z and Isabelle.

Initially the shallow embedding of *Circus* in Z is introduced. The model of a *Circus* program is a Z specification, the model of a process is a Z specification, and the model of an action is a schema. However, this semantics fails to prove the properties about *Circus* itself such as the refinement laws [73]. In order to overcome this limitation, a deep embedding of *Circus* in Z is proposed in [72] and [73]. With this new denotational semantics, it is able to describe the properties about *Circus* itself. Furthermore, based on this new semantics, the syntax and the semantics of *Circus* are mechanised in Z, as well as the *Circus* refinement laws.

The semantics of actions in *Circus* is characterised by three theorems from the paper [73]: Theorem 4.1, Theorem 4.2, and Theorem 4.3. They state that every *Circus* action is

- ***R*** healthy,
- ***CSP1***, ***CSP2***, and ***CSP3*** healthy,
- ***C1***, ***C2***, and ***C3*** healthy,

where ***C1***, ***C2***, and ***C3*** are given in Table 2.1. Actually ***C1*** and ***C2*** are corresponding maps of ***CSP4*** and ***CSP5*** in CSP to *Circus* respectively.

Furthermore, the operational semantics of *Circus* is defined in [44], [74] and [75], and it is also based on the alphabetised relations of UTP and expressed as Plotkin's SOS, or Structured Operational Semantics [76], style for the imperative and behavioural features of *Circus*. Frietas [44] has formalised the underlying automata theory and its properties in the theorem prover, Z/Eves.

2.2.3 Refinement Calculus of *Circus*

The support of refinement calculus is a key aspect which differentiates *Circus* from other combined formal languages. It is based on the UTP's correctness by logical implication to introduce the program from the specification. In the refinement strategy [64] of *Circus*, action refinement is the basic notion of refinement and covers behavioural aspects only. For processes, it must consider both data and behavioural aspects. Data refinement techniques, forward simulation and backward simulation [63], are used to refine the data aspect of a process. And then the refinement of the behavioural aspect of a process is based on the refinement of its main action. A simple reactive buffer [64] and a more complex industrial scale fire control system [35] are two typical cases that use the refinement calculus for development.

2.3 A *Circus* Example

In this section, a *Circus* model of the ESEL specification is shown below as an example to illustrate the basic structure and syntax of *Circus*. The detailed requirements of the ESEL system are given in Section 7.2.1. Particularly, the *Circus* model described below is also given in Appendix H.2. However, the only difference is the display of the model. In this section, the process *Controller* is displayed in multiple environment, therefore easily explained, while *Controller* in Appendix H.2 is displayed in a single environment without any description.

2.3.1 ESELHeader

The *ESELHeader* section is common for the three models of ESEL: Specification, System one (Section 7.2.3), and System two (Section 7.2.4). It is a parent section of each model to introduce all basic definitions.

ESELHeader includes the standard *Circus* toolkit, that will recursively include other Z toolkits, as its parent.

$$\textbf{section}\ \mathit{ESELHeader}\ \textbf{parents}\ \mathit{circus\_toolkit}$$

At first, two constants are defined. MAX_ESEL and MAX_PID stand for maximum number of displays and maximum number of product categories (or, products for short) in the system separately.

$$\begin{array}{|l}
MAX\_ESEL : \mathbb{N} \\
MAX\_PID : \mathbb{N}
\end{array}$$

Then all displays and products are identified by a tag plus a unique number which are defined in the free types $ESID$ and PID below where the constructors ES and PD are the tags for displays and products. For instance, number ten of the display is given $ES\,10$ or $ES\,(10)$.

$$\begin{array}{l}
ESID ::= ES\langle\!\langle 1 \mathinner{\ldotp\ldotp} MAX\_ESEL \rangle\!\rangle \\
PID ::= PD\langle\!\langle 1 \mathinner{\ldotp\ldotp} MAX\_PID \rangle\!\rangle
\end{array}$$

The type of product price is defined as an abbreviation to natural numbers $\mathbb{N}$.

$$Price == \mathbb{N}$$

The unit response is defined as a free type with two constants: uok and $ufail$.

$$UStatus ::= uok \mid ufail$$

The response from this program to the environment is a set of product identities of which the price is not updated successfully due to 1) no linked ESEL ID to the product or 2) failed update to its linked ESEL. The first reason is given the status constant NA and the second is provided with the constructor $fail\langle\!\langle ESID \rangle\!\rangle$.

$$FStatus ::= fail\langle\!\langle ESID \rangle\!\rangle \mid NA$$

Two channels are provided to update the map from ESEL ID to product ID. *updateallmap* will clear all stored map and use the input map as new map, while *updatemap* just updates a partial map. In this map, one ESEL can be linked to up to one product. However, one product may associate with multiple ESELs.

$$\begin{array}{l}
\textbf{channel}\ updateallmap : ESID \pfun PID \\
\textbf{channel}\ updatemap : ESID \pfun PID
\end{array}$$

Similarly, two channels are provided to update the price information. *updateallprice* will clear all price information and use the input price information as new price, while *updateprice* just updates price partially.

$$\begin{array}{l}
\textbf{channel}\ updateallprice : PID \pfun Price \\
\textbf{channel}\ updateprice : PID \pfun Price
\end{array}$$

The *update* channel gives a signal to the program to start update process.

$$\textbf{channel}\ update$$

The *failures* channel returns all products, which fail to update, and related error reasons after update. Since one product may associate with multiple displays, the return status is a power set of *FStatus* to denote which specific displays that the product links are updated unsuccessfully. But it is worth noting that *NA* and *fail* must not occur in a product's return set at the same time because they cannot be both no associate display and associate display update fail.

$\mathbf{channel}\ \ failures : PID \pfun \mathbf{P}\ FStatus$

The internal *resp* event is used to collect update responses from all displays and *terminate* event is for completing the collection.

$\mathbf{channel}\ \ resp : PID \times FStatus$
$\mathbf{channel}\ \ terminate$
$\mathbf{channelset}\ \ RespInterface == \{\!|\ resp, terminate\ |\!\}$

This *uupdate* event is to update one ESEL to the specific price, and *ures* for updating response from this ESEL. And *udisplay* is used to synchronise the show of price on all ESELs at the same time and *finishdisplay* is used to wait for display completion of all ESELs. That is the similar case for *uinit* and *ufinishinit* that are for initialisation synchronisation.

$\mathbf{channel}\ \ uupdate : ESID \times Price$
$\mathbf{channel}\ \ ures : ESID \times UStatus$
$\mathbf{channel}\ \ uinit, finishuinit$
$\mathbf{channel}\ \ udisplay, finishudisplay$

And *display* is used to synchronise the show of price on all gateways (or ESELs) at the same time and *finishdisplay* is used to wait for display completion of all gateways (or ESELs). That is the similar case for *init* and *finishinit* that are for initialisation synchronisation.

$\mathbf{channel}\ \ init, finishinit$
$\mathbf{channel}\ \ display, finishdisplay$

The channels below are for communication between the ESEL system and displays. The *write* event writes price to a display, and the *read* event reads price from the display. *ondisplay* turns on the related display and *offdisplay* turns off it conversely.

$\mathbf{channel}\ \ write : ESID \times Price$
$\mathbf{channel}\ \ read : ESID \times Price$
$\mathbf{channel}\ \ ondisplay : ESID$
$\mathbf{channel}\ \ offdisplay : ESID$

2.3.2 ESEL Specification

For the specification, the ESEL system is implemented by a central controller which consists of data modules, update modules, and response collection module as shown in Figure 2.1. Top three elements—*ESEL map*, *Price map*, and *Response*—keep the map between ESELs and products, the map between products and price, and the response respectively. Bottom n elements from U_1 to U_n are interleaved together, and each unit is responsible for the update of one corresponding display. Furthermore, each unit can read *ESEL map* and *Price map* but will not update them, and the update result from each unit is sent to the response collection module, *Collector*, which collects all responses and updates *Response*. With this architecture, all displays can be updated at the same time because of interleaved update units ($U_1, .., U_n$), and *Response* can be updated only by *Collector* and therefore it is not necessary to introduce a lock.

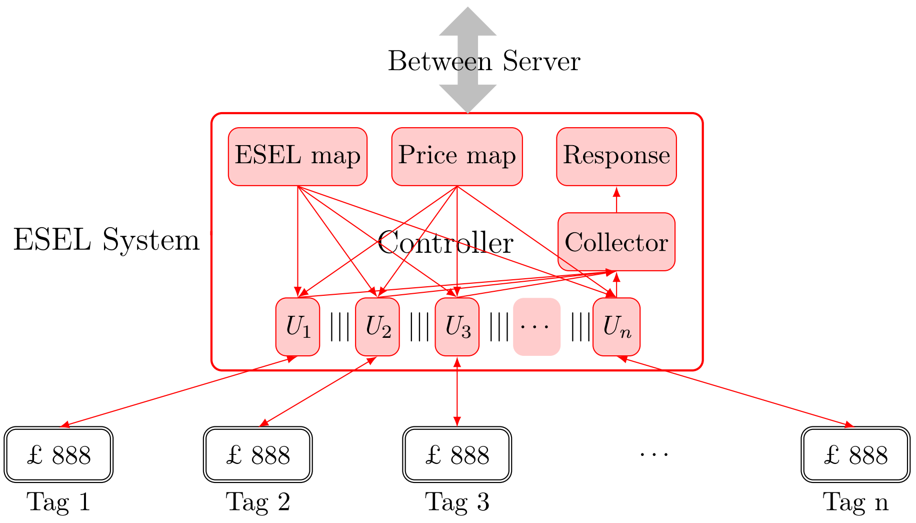

Figure 2.1: ESEL Specification Diagram

The *Circus* model of the specification is displayed below.

At first, *ESELSpec*, the name of the specification section, has *ESELHeader* as its parent. Consequently, it can use all definitions from *ESELHeader*, as well as the parents of *ESELHeader*, standard toolkits.

$$\textbf{section}\ \ ESELSpec\ \textbf{parents}\ \ ESELHeader$$

2.3.2.1 Controller Process

The process for overall control of the system, named *Controller*, is defined as an explicitly defined process.

$$\textbf{process}\ \ Controller \mathrel{\widehat=} \textbf{begin}$$

Controller has three state components: *pumap* for mapping from displays to products, *ppmap* for mapping from products to their price, and *response* for the response of one update to the environment.

$$\begin{aligned}&\textbf{state}\ \ State == [\, pumap : ESID \mathrel{+\!\!\!\!\!\twoheadrightarrow} PID\ ;\ ppmap : PID \mathrel{+\!\!\!\!\!\twoheadrightarrow} Price;\\ &\qquad response : PID \mathrel{+\!\!\!\!\!\twoheadrightarrow} (\mathbf{P}\ \ FStatus)\,]\end{aligned}$$

Initially, these three state components all are empty.

$$Init == [\,(State)' \mid pumap' = \varnothing \land ppmap' = \varnothing \land response' = \varnothing\,]$$

The *UpdateMap* schema partially updates the map between displays and products according to the input map, while the *UpdateAllMap* schema discards stored map and uses new input map as *pumap*.

$$\begin{aligned}&UpdateMap == [\,\Delta State\ ;\ map? : ESID \mathrel{+\!\!\!\!\!\twoheadrightarrow} PID \mid\\ &\qquad pumap' = pumap \oplus map? \land ppmap' = ppmap \land response' = response\,]\\ &UpdateAllMap == [\,\Delta State\ ;\ map? : ESID \mathrel{+\!\!\!\!\!\twoheadrightarrow} PID \mid\\ &\qquad pumap' = map? \land ppmap' = ppmap \land response' = response\,]\end{aligned}$$

The *NewPrice* updates a part of price information stored, while the *AllNewPrice* discards all price information stored and uses input price as *ppmap*.

$$\begin{aligned}&NewPrice == [\,\Delta State\ ;\ price? : PID \mathrel{+\!\!\!\!\!\twoheadrightarrow} Price \mid\\ &\qquad ppmap' = ppmap \oplus price? \land pumap' = pumap \land response' = response\,]\\ &AllNewPrice == [\,\Delta State\ ;\ price? : PID \mathrel{+\!\!\!\!\!\twoheadrightarrow} Price \mid\\ &\qquad ppmap' = price? \land pumap' = pumap \land response' = response\,]\end{aligned}$$

AUpdatemap is an action defined to update the map from displays to products: either partially by *updatemap* or completely by *updateallmap* event.

$$\begin{aligned}&AUpdatemap \mathrel{\widehat=} updatemap?map \rightarrow \left(UpdateMap\right)\\ &\quad \Box\ updateallmap?map \rightarrow \left(UpdateAllMap\right)\end{aligned}$$

Similarly, *ANewPrice* is an action defined to update the map from products to price: either a partial update by *updateprice* or a complete update by *updateallprice*.

$$\begin{aligned}&ANewPrice \mathrel{\widehat=} updateprice?price \rightarrow \left(NewPrice\right)\\ &\quad \Box\ updateallprice?price \rightarrow \left(AllNewPrice\right)\end{aligned}$$

A parametrised action, *AUpdateUnitPrice*, is given to update the price (specified indirectly by the formal *pid* parameter) to a display (given by the formal *uid* parameter). It sends the price to the specified display by the *write* event, and then read back the price from the display by the *read* event. If the write price matchs with the read price, then the update is successful. Otherwise, it fails (*ufail*) and sends the result to the response collection action *CollectResp*, then terminates.

$$\begin{aligned}&AUpdateUnitPrice \mathrel{\widehat=} uid : ESID\ ; pid : PID \bullet\\ &\quad write.uid.(ppmap\ pid) \rightarrow read.uid?y \rightarrow\\ &\quad ((y = (ppmap\ pid))\ \&\ \mathbf{Skip}\\ &\quad \Box\ (y \neq (ppmap\ pid))\ \&\ resp.pid.(fail\ uid) \rightarrow \mathbf{Skip})\end{aligned}$$

The parametrised action *AUpdateProductUnits* aims to update one product's price specified by the formal *pid* parameter in case the product has associated displays. Since one product may have more than one associated displays, this action updates the product's price to all associated displays. Furthermore, the update to each display is independent. Therefore, they are combined together into an interleaving. It is worth noting that each *AUpdateUnitPrice* action will not update state and local variables and thus its name set is empty.

$$\begin{aligned}&AUpdateProductUnits \mathrel{\widehat=} pid : PID \bullet\\ &\quad (\mathop{|||}\ uid : (\operatorname{dom}(pumap \rhd \{pid\}))\ \llbracket \varnothing \rrbracket \bullet AUpdateUnitPrice(uid, pid))\end{aligned}$$

Otherwise, if the product has not been allocated the corresponding displays, it sends back a response to state this error *NA*. The behaviour is defined in the *AUpdateNoUnit* action.

$$AUpdateNoUnit \mathrel{\widehat=} pid : PID \bullet resp.pid.NA \rightarrow \mathbf{Skip}$$

The behaviour of the price update for a product given in *pid* is the update of product either with associated displays, in the guarded *AUpdateProductUnits*, or without associated displays, in the guarded *AUpdateNoUnit*.

$$\begin{aligned}&AUpdateProduct \mathrel{\widehat=} pid : PID \bullet\\ &\quad\ \ (pid \in \operatorname{ran} pumap)\ \&\ AUpdateProductUnits(pid)\\ &\quad \Box\ (pid \notin \operatorname{ran} pumap)\ \&\ AUpdateNoUnit(pid)\end{aligned}$$

Then the update of all products is given in the action *AUpdateProducts*. At first, it is an interleave of all updates of the products which have associated price, then follows a *terminate* event to finish the update.

$$\begin{aligned}&AUpdateProducts \mathrel{\widehat=} ((\mathop{|||}\ pid : (\operatorname{dom} ppmap)\ \llbracket \varnothing \rrbracket \bullet AUpdateProduct(pid))\\ &\quad ; terminate \rightarrow \mathbf{Skip})\end{aligned}$$

The *AddOneFailure* schema adds one failure (either update failure or no associate failure) for a product to the state component *response*.

$$\begin{array}{l}
AddOneFailure == [\,\Delta State\,;\, pid? : PID\,;\, fst? : FStatus \mid \\
\quad (pid? \in \operatorname{dom} response \Rightarrow \\
\quad\quad response' = response \oplus \{pid? \mapsto (response(pid?) \cup \{fst?\})\}) \land \\
\quad (pid? \notin \operatorname{dom} response \Rightarrow \\
\quad\quad response' = response \cup \{pid? \mapsto \{fst?\}\}) \land \\
\quad ppmap' = ppmap \land pumap' = pumap\,]
\end{array}$$

The *CollectResp* action is to collect responses from all units and write them into the *response* variable by the *AddOneFailure* schema expression. It recursively waits for the response from the units, or terminates if required.

$$\begin{array}{l}
CollectResp \mathrel{\widehat=} \mu X \bullet \\
\quad ((resp?pid?fst \to \big(AddOneFailure\big)\,;\, X) \Box\ terminate \to \mathbf{Skip})
\end{array}$$

Then update of all products and response collection behaviours are put together into *AUpdateResp* action. It is a parallel composition of *AUpdateProducts* and *CollectResp* actions and they are synchronised on *resp* and *terminate* events. Furthermore, the left action *AUpdateProducts* will not update state variables (its name set is empty set) while the right action *CollectResp* will update *response* (its name set has only one variable *response*). Finally, these internal events are hidden.

$$\begin{array}{l}
AUpdateResp \mathrel{\widehat=} \\
\quad (AUpdateProducts \llbracket \varnothing \mid RespInterface \mid \{response\} \rrbracket\ CollectResp) \\
\quad \setminus RespInterface
\end{array}$$

All displays will synchronise on the *display* event to show the price at the same time, which is defined in *ADisplay*. Whether a display should be turned on or off is decided based on the logic below.

- If the display is not mapped to a product, then turn it off.
- Otherwise, if the display linked product is not to be updated, then turn it off.
- Otherwise, if the display has been written the price successfully, then turn it on.
- Otherwise, then turn it off.

$$\begin{array}{l}
ADisplay \mathrel{\widehat=} \\
\quad (\llbracket \{\!|\ display\ |\!\} \rrbracket\ uid : ESID \bullet \llbracket \varnothing \rrbracket\ display \to (\\
\quad\quad \mathbf{if}\ \ uid \notin \operatorname{dom} pumap \longrightarrow offdisplay.uid \to \mathbf{Skip} \\
\quad\quad [\!]\ uid \in \operatorname{dom} pumap \longrightarrow \\
\quad\quad\quad \mathbf{if}\ \ pumap(uid) \notin \operatorname{dom} ppmap \longrightarrow offdisplay.uid \to \mathbf{Skip} \\
\quad\quad\quad [\!]\ pumap(uid) \in \operatorname{dom} ppmap \longrightarrow \\
\quad\quad\quad\quad \mathbf{if}\ \ pumap(uid) \notin \operatorname{dom} response \longrightarrow \\
\quad\quad\quad\quad\quad ondisplay.uid \to \mathbf{Skip} \\
\quad\quad\quad\quad [\!]\ pumap(uid) \in \operatorname{dom} response \longrightarrow \\
\quad\quad\quad\quad\quad \mathbf{if}\ (fail\ uid) \notin response(pumap(uid)) \longrightarrow \\
\quad\quad\quad\quad\quad\quad ondisplay.uid \to \mathbf{Skip} \\
\quad\quad\quad\quad\quad [\!]\ (fail\ uid) \in response(pumap(uid)) \longrightarrow \\
\quad\quad\quad\quad\quad\quad offdisplay.uid \to \mathbf{Skip} \\
\quad\quad\quad\quad\quad \mathbf{fi} \\
\quad\quad\quad\quad \mathbf{fi} \\
\quad\quad\quad \mathbf{fi} \\
\quad\quad \mathbf{fi} \\
\quad)) \setminus \{\!|\ display\ |\!\}
\end{array}$$

The overall price update action is given in $AUpdatePrice$, which accepts an *update* event from its environment, then clears *response*, updates the price, sends the *display* event to make all ESELs show their price at the same time, then feeds back the response to the environment.

$$\begin{array}{l} AUpdatePrice \mathrel{\widehat=} update \rightarrow response := \varnothing; \\ \quad AUpdateResp \mathbin{;} ADisplay \mathbin{;} failures.response \rightarrow \mathbf{Skip} \end{array}$$

Initially, state components are cleared and all displays are turned off.

$$AInit \mathrel{\widehat=} \left(Init\right) \mathbin{;} (\mathop{|||}\ u : ESID \ \llbracket \varnothing \rrbracket \bullet offdisplay.u \rightarrow \mathbf{Skip})$$

The overall behaviour of the *Controller* process is given by its main action. It initializes at first, then repeatedly provides the update of stored maps, or the update of price to its environment.

$$\bullet\ AInit \mathbin{;} (\mu X \bullet (AUpdatemap \mathbin{\Box} ANewPrice \mathbin{\Box} AUpdatePrice) \mathbin{;} X)$$

end

2.3.2.2 System

The ESEL Specification is simply the *Controller* process.

$$\mathbf{process}\ ESELSpec \mathrel{\widehat=} Controller$$

2.4 Combination of CSP and B

$CSP \parallel B$ is a combination of CSP and B which adds behavioural specification to the state-based B machine. The B method characterises abstract state, operations with respect to their enabling conditions, and their effect on abstract state, while CSP specifies overall system behaviour. But different from *Circus*, the CSP specification and the B machine in $CSP \parallel B$ are always orthogonal. They are individually complete specifications and can be checked separately.

Semantically, the combination of CSP and B works in such ways in terms of CSP: the B machine, which has operations Ops ($\{Op_1, \ldots, Op_n\}$) and variables $Vars$ ($\{v_1, \ldots, v_m\}$), is regarded as a process in CSP with events ops ($\{op_1, op_2, \ldots, op_n\}$, the event name is the same as the operation name) available for its environment; CSP and B are composed together by a generalised parallel composition in CSP. B and CSP synchronise on these common events ops. For an event op_i in common events ops, if it is allowed at the same time by both the B machine and the CSP specification, then it can make progress. After that, the state $Vars$ of B machine is updated or accessed by op_i related operation Op_i. Otherwise, if it is not allowed by either B machine or CSP, neither of them can progress. In addition, for the event only specified in CSP and not having a corresponding operation in B, it engages independently. But for the operation only specified in B and not in CSP, it is prevented from executing.

For example, a B machine, named B_{mach}, has the state B_{state} and a set of operations $OPs = \{Op_1, Op_2, \ldots Op_n\}$. The behaviour of this B machine is treated as a CSP process B_{proc} shown in Equation 2.1 with the corresponding events $ops = \{op_1, op_2, \ldots op_n\}$, and the name of event op_1 is the same as the name of the operation Op_1. (Notes: the upper case Op denotes operation name in B and the lower case op for corresponding event name in CSP.)

$$B_{proc} \mathrel{\widehat=} \mu\, X \bullet \left(\begin{array}{ll} & ((op_1 \rightarrow X) \lhd enabled\,(Op_1) \rhd STOP) \\ \Box & \ldots \\ \Box & ((op_n \rightarrow X) \lhd enabled\,(Op_n) \rhd STOP) \end{array} \right) \tag{2.1}$$

An additional restriction is that B_{proc} cannot always engage in parallel and it only happens when the precondition of the operation is met or the operation is *enabled* ($enabled(Op)$ is true). If the operation is enabled, the corresponding event op is available to synchronise with CSP process, and after synchronisation the state may be updated or accessed by the corresponding operation Op. After that, the enable condition for each operation will be calculated again, since the state may be changed. If the operation is not enabled, then the corresponding event is not available for the environment. It's impossible for this operation to be executed. If none of the operations are enabled, then the whole B machine is deadlocked. Consequently no state can be changed and accessed.

The overall combination of CSP and B specification is regarded as the parallel composition of B_{proc} and CSP plus further hiding of the events between them, which is shown in 2.2:

$$CSP \parallel B \mathrel{\widehat=} \left(CSP \;\; {}_{C}\!\underset{ops}{\parallel}_{B} \; B_{proc} \right) \setminus ops \tag{2.2}$$

where ${}_{C}\!\parallel_{B}$ is a new defined parallel operation for the $CSP \parallel B$ model only.

Definition 2.4.1 (Parallel Composition for the combination of CSP and B). ${}_{C}\!\parallel_{B}$ *is a parallel composition in terms of CSP. The only difference from the general parallel composition* $\parallel$ *is that the process* P *in the* C *part is a controller and gives the whole behaviour of the parallel composition. The events in the* B *part are only available to its environment if* P *allows them. From other aspects,* ${}_{C}\!\parallel_{B}$ *is exactly the same.*

${}_{B}\!\parallel_{C}$ *is a similar parallel composition but* $B \;\, {}_{B}\!\parallel_{C}\, P = P \;\, {}_{C}\!\parallel_{B}\, B$.

A number of laws are defined below to facilitate the reasoning of $CSP \parallel B$ programs. It is worth noting that basic processes in CSP denote $SKIP$, $STOP$, and **div**. However, in *Circus*, a basic process denotes an explicitly defined process. In addition, we use the equation

$$(P) \;\; {}_{C}\!\underset{ops}{\parallel}_{B} \left({B_{proc}}_{v}^{v'} \right)$$

to represent a $CSP \parallel B$ program that has before state v and after state v', where v denotes a set of all undashed state variables and v' for a set of all dashed state variables. This equation means that the program has input state v and after P terminates the state has been changed to v'.

Another notation is introduced in the RHS of laws is

$$P \wedge_R pred \mathrel{\widehat=} \boldsymbol{R3}\,(P \wedge pred)$$

where P is a process in terms of $CSP \parallel B$, $pred$ is a predicate, and R in $\wedge_R$ denotes the composed healthy condition $\boldsymbol{R}$ for the reactive processes theory in UTP. The reason why $\boldsymbol{R3}$ is introduced into RHS is that 1) $CSP \parallel B$ is defined in terms of CSP and every CSP process is reactive and the $\boldsymbol{R}$-image of design according to Theorem 2.1.1, 2) however $pred$ in RHS alone is an alphabetised relation and is not reactive, and it will cause a sequential composition problem illustrated below.

Provided P and Q are processes in $CSP \parallel B$, therefore both of them are ***R3***-healthy. Sequence composition will result in

$$
\begin{array}{l}
(P \wedge pred_1) ; (Q \wedge pred_2) \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P \wedge pred_1)\, [okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \\ \qquad \wedge\ (Q \wedge pred_2)\, [okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \end{array} \right) \\
\hfill [\text{Sequence as relational composition}] \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \wedge pred_1[v_0/v']) \\ \qquad \wedge\ (Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \wedge pred_2[v_0/v]) \end{array} \right) \\
\hfill [\text{Substitution and } pred \text{ only has state variables } v] \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v']) \\ \qquad \wedge\ (Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v]) \\ \qquad \wedge\ (pred_1[v_0/v'] \wedge pred_2[v_0/v]) \end{array} \right) \\
\hfill [\text{Predicate Calculus}] \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v']) \\ \qquad \wedge\ (I\!I_{rea} \lhd wait \rhd Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v]) \\ \qquad \wedge\ (pred_1[v_0/v'] \wedge pred_2[v_0/v]) \end{array} \right) \\
\hfill [Q \text{ is } \textbf{\textit{R3}}\text{-healthy}] \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad \left(\begin{array}{l} P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \\ \wedge I\!I_{rea} \wedge (pred_1[v_0/v'] \wedge pred_2[v_0/v]) \end{array} \right) \\ \qquad \lhd wait \rhd \\ \qquad \left(\begin{array}{l} P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \\ \wedge\ Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \\ \wedge\ (pred_1[v_0/v'] \wedge pred_2[v_0/v]) \end{array} \right) \end{array} \right) \\
\hfill [\text{Conditional and Predicate Calculus}]
\end{array}
$$

So this result states that even Q has not started ($wait = false$), the predicate $pred_2$ has been established and state might be changed. It is opposite to what we expect: $pred_2$ is established only after Q has terminated. In order to overcome this problem, we introduce ***R3***. Finally,

$$
\begin{array}{l}
(P \wedge_R pred_1) ; (Q \wedge_R pred_2) \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P \wedge_R pred_1)\, [okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \\ \qquad \wedge\ (Q \wedge_R pred_2)\, [okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \end{array} \right) \\
\hfill [\text{Sequence as relational composition}] \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \wedge_R pred_1[v_0/v']) \\ \qquad \wedge\ (Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \wedge_R pred_2[v_0/v]) \end{array} \right) \\
\hfill [\text{Substitution and } pred \text{ only has state variables } v] \\
= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \qquad (P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \wedge_R pred_1[v_0/v']) \wedge \\ \qquad \left(\begin{array}{l} I\!I_{rea} \\ \lhd wait \rhd \\ (Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \wedge pred_2[v_0/v]) \end{array} \right) \end{array} \right) \\
\hfill [\text{Defintion of } \wedge_R\text{: } \textbf{\textit{R3}}\text{-healthy}]
\end{array}
$$

$$= \left(\begin{array}{l} \exists\, okay_0, wait_0, tr_0, ref_0, v_0 \bullet \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \\ \wedge_R\ pred_1[v_0/v'] \end{array} \right) \\ \wedge\, I\!I_{rea} \end{array} \right) \\ \quad \lhd wait \rhd \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} P[okay_0, wait_0, tr_0, ref_0, v_0 / okay', wait', tr', ref', v'] \\ \wedge_R\ pred_1[v_0/v'] \end{array} \right) \\ \wedge \left(\begin{array}{l} Q[okay_0, wait_0, tr_0, ref_0, v_0 / okay, wait, tr, ref, v] \\ \wedge\ pred_2[v_0/v] \end{array} \right) \end{array} \right) \end{array} \right)$$

[Conditional and Predicate Calculus]

Therefore, the sequence problem is overcome.

Furthermore, for $(P \wedge_R pred)$, it is simply both ***R1*** and ***R2*** healthy. Finally, it is ***R***-healthy and we can say the RHS of the laws below is also reactive.

Law 2.4.1 (Basic Processes).

$$(SKIP)\ \ {}_{C}\|_{B}^{\ } {}_{ops} \left(B_{proc}{}_{v}^{v'} \right) = SKIP \wedge_R (v' = v)$$

$$(STOP)\ \ {}_{C}\|_{B}^{\ } {}_{ops} \left(B_{proc}{}_{v}^{v'} \right) = STOP \wedge_R (v' = v)$$

$$(\mathbf{div})\ \ {}_{C}\|_{B}^{\ } {}_{ops} \left(B_{proc}{}_{v}^{v'} \right) = \mathbf{div} \wedge_R \mathbf{true}$$

This law states that the behaviour is the same as that in CSP and the state in B remains unchanged for termination and deadlock, but unconstrained for divergence.

Law 2.4.2 (Prefix (Independent Event)). *For the channel c that is in CSP only, it can engage independently and the state in B remains unchanged.*

$$(c \to P)\ \ \underset{ops}{{}_{C}\|_{B}} \left(B_{proc}{}_{v}^{v'} \right) = \exists\, v_0 \bullet \left(c \to \left(P\ \ \underset{ops}{{}_{C}\|_{B}} \left(B_{proc}{}_{v_0}^{v'} \right) \right) \right) \wedge_R (v_0 = v)$$

Law 2.4.3 (Prefix (Synchronisation Event)). *Provided the precondition of the operation op, pre (op), in B holds or the operation op is enabled, then the op event is available to its environment.*

$$(op \to P)\ \ \underset{ops}{{}_{C}\|_{B}} \left(\mu X \bullet \left(\begin{array}{l} ((op \to X) \lessdot enabled\,(Op) \gtrdot STOP) \\ \ldots \end{array} \right) \right)_{v}^{v'}$$

$$= \exists\, v_0 \bullet \left(\begin{array}{l} \left(\begin{array}{l} op \to \\ \left(\begin{array}{l} P\ \ \underset{ops}{{}_{C}\|_{B}} \\ \left(\mu X \bullet \left(\begin{array}{l} ((op \to X) \lessdot enabled\,(Op) \gtrdot STOP) \\ \ldots \end{array} \right) \right)_{v_0}^{v'} \end{array} \right) \end{array} \right) \\ \wedge_R\ post\,(Op)\,[v_0/v'] \end{array} \right)$$

After the engagement of the op event, the postcondition of the op operation, post (op), is established. post (op) $[v_0/v']$ means all free occurrences of dashed variables v' in post (op) are substituted by v_0 and sequentially v_0 is the input state of the following parallel composition of the process P and the B machine with newly updated state. Therefore the state is changed. It is worth noting that the two occurrences of v' in the equation above have different scope. The first v' denotes the after state of the parallel composition, while the second one represents the dashed variables in the post condition of op.

Otherwise, if the precondition of the operation op in B does not hold or the operation op is not enabled, then the op event is not available to its environment. The state is not changed.

$$\begin{array}{l}(op \rightarrow P) \underset{ops}{{}_C\|_B} \left(\mu X \bullet \left(\begin{array}{l} ((op \rightarrow X) \lhd enabled\,(Op) \rhd STOP) \\ \ldots \end{array} \right) \right)_v^{v'} \\ = STOP \wedge_R (v' = v)\end{array}$$

Law 2.4.4 (Sequential Composition).

$$(P\,;\,Q) \underset{ops}{{}_C\|_B} \left(B_{proc}{}_v^{v'}\right) = \exists\, v_0 \bullet \left(\left(P \underset{ops}{{}_C\|_B} \left(B_{proc}{}_v^{v_0}\right) \right) ; \left(Q \underset{ops}{{}_C\|_B} \left(B_{proc}{}_{v_0}^{v'}\right) \right) \right)$$

2.4.1 Semantic Considerations

Various semantics have been presented for combinations of CSP and B.

One approach, as presented along with csp2B [77], is to combine CSP-like descriptions with standards B machines together to allow CSP notations to put constraints on the order in which the operations in B machines can occur. Using csp2B, a model in a pure CSP specification or a combination of CSP-like descriptions and B machines can be translated to a model in pure B machines. Butler [77] uses an operational approach to giving the semantics to the CSP and B combination and justifying the translation as well.

Schneider and Treharne [15] extended Morgan's CSP semantics for event systems [78], which uses weakest precondition notations over action systems to express traces, failures and divergences of CSP, to B machines and introduced an additional stable failures semantics to B machines. Since both CSP and B have the traces model, the stable failures model, and the failures-divergences model now, the semantics of the combination of CSP and B, $CSP \parallel B$, can be analysed by using these models together.

Another approach to combining CSP and B, which is the approach used in ProB, is through the combination of operational semantics of both CSP and B [79]. The state of a combined CSP and B specification is a pair from the state of the B machine to the state of the CSP process. The transition from an old state to a new state in the combined model is labelled with channel events which link CSP events to B operations. The CSP events and the B operations are the labelled transitions in the transition systems of CSP and B respectively.

Since the denotational semantics of *Circus* is based on UTP, our treatment of the semantics of $CSP \parallel B$ is also specified in UTP. The other semantics of combining CSP and B facilitate our understanding of $CSP \parallel B$ in ProB as well as our definition of UTP semantics for $CSP \parallel B$.

Chapter 3

Model Checking Approaches

In the beginning of this chapter, we discuss the difficulties of developing a model checker for *Circus* from scratch. Then Section 3.2 gives a brief introduction of current model checking approaches for *Circus*. After that, Section 3.3 presents our solution in detail. Particularly, the differences from other solutions and the considerations of our approach are given in the end.

3.1 Difficulties

The difficulties of developing a model checker for state-rich *Circus* language lie in several factors.

Model checking is a technique to use an exhaustive search of the system's state space to check whether the system model in a formal language satisfies a desired specification.

The state explosion problem [32, 33, 71] is the biggest challenge for model checkers and well-known to model checking communities. The problem originates from the fact that the size of the state space grows exponentially with the number of processes if it is explicitly enumerated. For instance, if a system is composed of m processes and each process has n states, then the whole system has n^m states. Even for a small system, it is still hard or impossible to represent and search this state space explicitly.

Roscoe [71, Chapter 17] views the problems as two sorts. The first one is a concrete system with large data-types, many parallel processes, or both of them. These concrete systems may have large finite state spaces or infinite state spaces due to infinite types or unlimited number of parallel processes. And the second one is a family of systems, usually infinitely many, by parametrisation. The main challenge for concrete systems is the growing state space, while for the parametrised systems there rises another problem about the proof of an infinite number of concrete systems at the same time. This is known as the *parametrised verification problem.*

Circus is a language designed for the state-rich concurrent systems. For these systems, the size of their state spaces is intrinsically huge. This may be due to infinite data types, nondeterminism (from disjunction of state operations and internal choice), and unbounded number of parallel processes, which is the first sort of problem as described above. Or the family of concrete systems introduced by parametrisation and loose constants in *Circus*, which is the second sort of problem. A loose specification [63, p. 367] has a number of constants introduced with a range of values (maybe infinite like $\mathbb{N}$). Uninitialised state variables also results in this looseness. Finally, these loose constants are regards as the parameters to the specification, and consequently the model specifies a family of systems. Therefore, to cope with the state explosion problem for the systems specified in *Circus*, it is even harder.

Additionally, expressiveness power of *Circus*, such as rich notations and flexible user defined operators, makes the development of model checker for *Circus* very difficult in practice.

3.2 Current Approaches

A number of solutions have been proposed to model-check *Circus* models. Some of them are developed based on the operational semantics of *Circus* while others are merely transformed models to other formal languages and then use currently applicable model checkers for these languages to check translated models, which indirectly checks the original *Circus* models.

3.2.1 Operational Semantics Based

3.2.1.1 Model Checker Prototype By Freitas

Freitas proposed a model checker architecture in his PhD thesis [44, Figure 4.1]. The basic idea is similar to that of FDR to use refinement checking. Firstly, the syntax of a model is transformed into internal transition system by a compiler. The transition system used for *Circus* is a predicate transition system (PTS) [44, p. 84] which is a variation of the labelled transition system (LTS). Then the specification side of refinement is normalised to eliminate nondeterminism and internal transitions. Finally, an algorithm is used to check the transition system of the implementation against the normalised specification to determine if the implementation is a correct refinement of the specification. Additionally, a theorem prover is integrated into the proposed model checker. The work finally results in a parser and type checker for *Circus* in CZT [80] project and a prototype model checker [44, Chapter 5] developed in Java.

3.2.1.2 Model Checker on FORMULA

Another operational semantics based model checker [47, 48] is built on Microsoft FORMULA framework [49] which is based on constraint logic programming (CLP) and satisfiability modulo theories (SMT) solving introduced from the Z3 solver [81]. This model checker can support both state-rich *Circus* and CML languages. Its rapid prototyping is very interesting. It takes 72 hours to build a semantically well founded model checker for *Circus*, following two months to learn FORMULA and eight months to build a reusable framework [47, 48].

The methodology of this work is to 1) use abstraction to get a semantically equal description in logic (FORMULA) from the input *Circus* or CML model, 2) convert the properties to be checked into this logic at the same time, then 3) combine them together to form a logic representation, and 4) finally feed it to the SMT solver to check if the desired properties hold in the input model. The goal of this model checker is to preserve the semantics and give correctness by construction instead of the optimal performance.

Based on this methodology, this model checker is able to check some basic properties, such as deadlock, divergence, and determinism for a *Circus* model, or application specific properties in the temporal logic. However, it is not capable of checking whether one *Circus* program is a correct refinement of another specification, that is, the lack of refinement model checking.

3.2.2 CSP$_M$ and FDR

As we know, two model checking solutions: the *link* [52, 53] and the *map* [54, 55], are proposed to link or map *Circus* to CSP (CSP$_M$) and then finally use FDR to model check the linked or mapped CSP specifications.

The *link* is based on the translation rules given in the work [35, Chapter 5] which translates *Circus* to Java with JCSP. Instead of Java and JCSP, the *link* translates both state and behavioural aspects of *Circus* to CSP$_M$. In addition, a translator prototype [53], which is an extension based on *JCircus* source code, is developed. The basic methodology behind this solution is to translate the state components of a process in *Circus* to the parameters list of a process in CSP. Obviously, the type information of state variables are lost after translation. In addition, it is not clear how to cope with schema expressions in this solution because schema expressions are natural in *Circus*.

The *map*, however, takes a different approach. It first transforms the state-rich *Circus* model into a stateless model using the memory model [59]. For each basic process, its state variables are modelled and kept in a *Circus* action *Memory*, which provides access and update of each state variable through internal events *mget* and *mset*. Then this *Memory* action is put in parallel with the main action of the process. By this way, the state schema is eliminated and consequently the process becomes stateless. Then after that, in its second step, the stateless *Circus* model is transformed to CSP_M. Finally, the CSP model is model-checked by FDR.

Both solutions have their main advantages to use the powerful model checker FDR that could work on an industry scale.

3.2.3 Refinement Calculus

CRefine [82] is a refinement tool and not a model checker. Its methodology to preserve correctness between refinements and the specification is through its well-proven refinement strategy and laws [35,64]. So from the specification, ***CRefine*** can guide the step-wise refinement according to the laws integrated. However, it cannot check properties for individual *Circus* model or refinement between different *Circus* models.

3.2.4 Implementation

There is one solution, *JCircus* [35, 50, 51, 56], that translates a concrete *Circus* model into Java. Fundamentally, it is not a model checker solution. Instead, it is an implementation of *Circus* in the programming language. By *JCircus*, the concrete and executable *Circus* model is translated to Java. The basic strategy is to translate the data part of *Circus* into Java directly and the behavioural part in CSP to JCSP which is an implementation of CSP in Java. Therefore, this tool requires the *Circus* models should be refined to an executable model before translation. To be integrated with ***CRefine***, this tool obviously has its value.

3.3 Our Approach

Our approach, model checking of *Circus* by linking to a combination of a state-based formalism and a process algebra, is proposed based on our understanding of the difficulties to develop a model checker for *Circus* from scratch, and the review of current tools.

Similar to the *map* and the *link*, our solution is based on the translation of *Circus* to another formal language too. But the main difference of our approach is to preserve the state part of *Circus*, which is described in Z, still in a state-based language instead of CSP_M. Therefore, the majority of the data-rich aspect of *Circus* will be kept.

Since *Circus* is a combination of Z and CSP, the translation of *Circus* to an integrated formal language which is composed of Z and CSP is obviously our first consideration. CSP-OZ [11] and CSP-Z [13, 14] are two examples. However, both of them use the similar strategy [14, 17, 19] to model-check their specifications: transformed to CSP_M and then checked by FDR. Therefore, compared to the direct translation of *Circus* to CSP_M, this method does not have any advantages. Finally, we choose $CSP \parallel B$ [15] as one of integrated formalisms to which *Circus* models are translated. $CSP \parallel B$ has its own model checker ProB [20] which supports the checking of deadlock and divergence as well as temporal logics formulas.

Table 3.1: Comparison of Various Tools in Model Checking Aspect

Item	FORMULA	***CRefine***	CSP_M (FDR)	Our Approach
		Overall		
Abstraction	Yes	Yes	No	Partial loss
Looseness	Yes	Yes	No	Partial support
Nondeterminism	Yes	Yes	Partially support	Yes
Performance	-	-	Good	Not good
		Properties		
Refinement	No	Yes	Yes	Yes (Traces and Failures)
Deadlock	Yes	-	Yes	Yes
Divergence	Yes	-	No	Yes
Deterministic	Yes	-	Yes	Yes
Temporal Logic	Yes	-	No	Yes
		Syntax and Semantics		
External Choice	Yes	Yes	Partially	Yes (Extension)
Parallel Composition and Interleaving	Yes	Yes	Yes	Yes (Extension)
		Miscellaneous		
Animation	No	No	Yes	Yes
Counterexample	-	-	Yes	Yes
Development of Translator	-	-	Complex	Less complex

3.4 Final Considerations

In this section we compare our approach with other solutions, and present our considerations to use this approach in detail.

The comparison of these solutions from model checking aspect is illustrated in Table 3.1. In the table, not all tools reviewed in Section 3.2 are listed. The reason to exclude Freitas's model checker prototype is because it does not get further development and consequently no applicable tool. And because *JCircus* is not a model checker solution, it is omitted as well. In the table, FORMULA denotes the model checker on FORMULA and CSP_M (FDR) denotes both the *link* and the *map* solutions. Since the FORMULA solution and ***CRefine*** are not a refinement model checker — however refinement checking is very important in our objectives, they are just displayed in the table but no many details given. Finally, the main solutions to be compared are the *link* and the *map*.

3.4.1 Overall

Abstraction is very important to *Circus*, which makes it suitable for modelling the state-rich systems in its specifications. For example, abstract data type in *Circus*, similar to Z, comprises a state schema to describe the state space, and a number of operations, each of them gives the relation between input and output variables, the state before operation, and the state after operation. Since CSP does not support abstract types such as $\mathbb{N}$, the translation of an abstract *Circus* specification into a concrete CSP specification will result in information loss. Our solution finally still uses a B machine to represent the state part, and consequently preserve the abstraction. However, if these abstract types are used in the behavioural part, our solution has the same issue as the translation to CSP. Eventually, our solution may have partial loss of abstraction.

Loose specifications, the specifications with global constants (may have uninitialized state variables too) to define a family of systems, are natural in *Circus*. But CSP does not support looseness. For our solution, if these constants are only used in the state part, they will be converted to B which supports loose constants. However, if they are used in the behavioural part too, we have to give a specific instantiation of all constants that are used in CSP. Finally, our solution partially supports looseness.

Nondeterminism in *Circus* is introduced by the disjunction of operations or internal choice. When it is translated into CSP, the nondeterminism from the disjunction of operations will be lost. Our solution translates the disjunction of operations into B and consequently keeps this nondeterminism.

In the end, ProB's performance and scalability are not as good as FDR because its underlying logic language (SICStus Prolog [83]) cannot support multithreading. This is the major advantage of the solutions that translate *Circus* to CSP when compared to our solution.

3.4.2 Properties

Refinement checking obviously is the most desired feature for a *Circus* model checker because *Circus* is characterised by its refinement calculus. For the solutions relying on FDR, they can use all refinement models in FDR, such as the traces model, the failures model, and the failures-divergences model. Our solution can check refinement in only two models: traces model and failures model.

For other typical properties (deadlock, divergence, and deterministic), the divergence checking of our solution is different. In *Circus*, in addition to divergence arising from the behavioural part in CSP notations, divergence may arise from the unsatisfied precondition of a schema expression as action, a specification statement, or an assumption, or from an alternation in which all guard conditions does not hold. The *map* restricts to divergence-free *Circus* models. In our solution, this divergence information is also captured (see Link Rule 31, 53, 55, and 56) in an explicit process **div** and it is checkable.

Furthermore, our solution is able to check other application specific properties specified in temporal logics: LTL and CTL [31]. Particularly, ProB supports formulas to specify past states and transitions in its extension of LTL, $LTL^{[e]}$ [84].

3.4.3 Syntax and Semantics

In *Circus*, external choice, parallel composition, and interleaving of actions have non-trivial difference from those in CSP.

For external choice, it could only be resolved by external events or termination, but not internal state changes. For instance,

$$(s := 1 \,;\, c.s \rightarrow \textbf{Skip}) \,\Box\, (s := 2 \,;\, d.s \rightarrow \textbf{Skip})$$

provides its environment with two events ($c.1$ and $d.2$). Only after the environment has made a choice, then the external choice is resolved. Upon resolution, the state variable s is updated correspondingly: if $c.1$ is engaged, then s is equal to 1; otherwise, it is 2. This semantics make it hard to be related to CSP notations. Due to this, the *map* limits the actions in an external choice to prefixed actions. That is the similar case as used in our solution in this document. We also restrict the actions to prefixed actions AA (see

Definition B.3.1) in R_{wrt} Rule 30. However, this limitation could be eliminated in our solution by introducing an extra notation in $CSP \parallel B$ to represent the corresponding external choice in *Circus*, then extending the ProB's kernel to support this lazy resolution of the external choice.

As to parallel composition and interleaving of actions, such as

$$(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$$
$$(A_1 \interleave[ns_1 \mid ns_2]\interleave A_2)$$

Circus introduces two disjoint name sets ns_1 and ns_2. Both A_1 and A_2 are allowed to access or modify all variables in scope, but only the changes made in A_1 to ns_1 and in A_2 to ns_2 will be kept after termination. The *map* rewrites parallel composition or interleaving to have an additional copy of local memory *MemoryMerge* for each action, then use a *Merge* process to merge two local memories into the main memory. In this thesis, we limit it to two situations: "Disjoint Variables in Scope" (Φ Rule 30) and "Disjoint Variables in Updating" (Φ Rule 31). It is also possible to support them fully by using a similar strategy. We can rewrite parallel composition and interleaving at first to declare two copies of temporary variables by variable block, and in each copy, every temporary variable has a corresponding variable in $scpV$, where $scpV$ is a function to get a set of all variables in scope in an action. Initially, these temporary variables are initialised to the values of their corresponding variables in $scpV$. Then in A_1 and A_2, instead of updating variables in $scpV$ directly, we update its temporary copy. After that, int the end of parallel composition and interleaving, only variables in ns_1 and ns_2 will be assigned with new values from their corresponding copies of temporal variables. However, it is difficult to verify this solution and implement it in the translator. Finally, we do not support this solution in this work.

3.4.4 Miscellaneous

Debugging plays a significant role in model checking. When an error arises during model checking, it is essential to provide a counterexample to give enough hints about the trace which leads to this error. In addition, the capability to animate *Circus* models is also very useful. It helps users or engineers to get a good understanding of systems in design in its early stage or demonstrate systems before being made. The animator of ProB is very straightforward and user-friendly.

Compared to the *link* and the *map* that translate *Circus* to CSP, our solution uses a different approach to $CSP \parallel B$. Therefore, the complexity of the translator development is different. Comparatively, our translator would be less complex because it does not need to translate all constructs in a state-rich *Circus* to CSP_M and most of them are still kept in the final resultant B machine.

3.4.5 Summary

Our approach to model-check *Circus* is presented based on an analysis of all these aspects. The major drawback of our approach is the performance concern which really relies on the possible improvement of ProB in the future. In case of performance improvement, our approach will be significantly valuable.

Chapter 4

Link

In this chapter, the link from *Circus* to $CSP \parallel B$ of our approach is defined. Our link definition is based on the semantics of both *Circus* and $CSP \parallel B$. The main objective of the link definition is to preserve the semantics of each construct in *Circus* after it is linked to $CSP \parallel B$. Its soundness is given in Chapter 5.

This chapter first shows overall definition of the link, its strategies, and decompositions. The link mainly consists of three decomposed functions: the rewrite function - R_{wrt}, the state part translation function - Ω, and the behavioural part translation function - Φ. The Ω function is decomposed further into three sub-functions — Ω_1, Ω_2, and Ω_3 — for different stages of the state translation. In the second part of this chapter, these functions are defined by a collection of rules for constructs from their domain. Basically, we give the rules to the general constructs at first, then to sections, expressions, given type paragraphs, axiomatic definition paragraphs, channel declarations, processes, and finally to actions. All these rules are defined individually for each construct of one function. To link a construct in *Circus* to the final $CSP \parallel B$ program, according to our link definition and decompositions, actually it is an application of multiple rules from different functions sequentially. Therefore, to link a specification by application of individual rules would be long and tedious. Eventually, we present the Link Rules in Appendix E, which link from *Circus* to the combination of *CSP* and *Z* by combining several individual rules in this chapter together for each construct in *Circus*. Consequently, when linking a *Circus* specification or construct, we can use Link Rules directly regardless of all internally individual rules.

4.1 Link Strategies and Functions

4.1.1 Overall Link Strategies

Our link from a *Circus* program to a $CSP \parallel B$ program is defined as a function Υ (pronounced "upsilon").

$$\mathsf{Circus} \xRightarrow{\Upsilon} CSP \parallel B$$

The overall link strategies of our approach are illustrated in Figure 4.1 and Figure 4.2, which are seen from the language view and the structural view respectively.

From the language view, *Circus* is a combination of Z, CSP, and guarded commands. The syntax of *Circus* is a free mixture of Z and CSP. The interaction between the state part and the behavioural part is not limited to schema expression as action, and it includes other actions, such as communications capable of accessing state variables directly, assignments to update or access state variables, specification statements to specify the state part, and guarded commands to evaluate state variables. The overall strategies of Υ from the language view are given below.

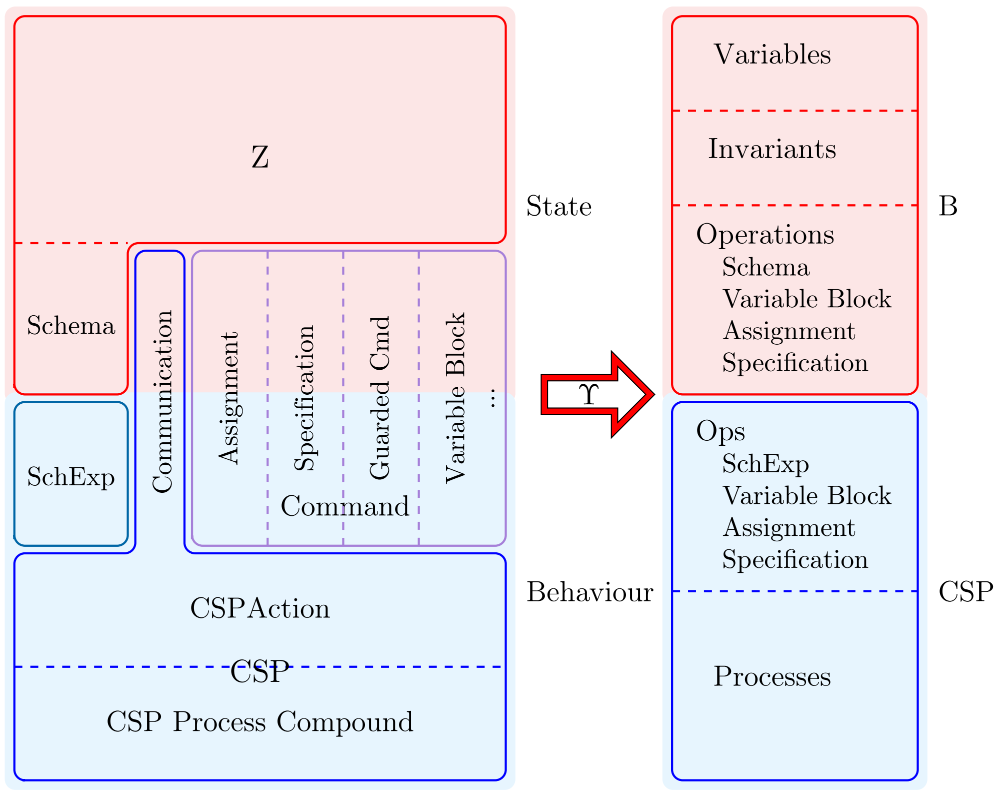

Figure 4.1: Link Relation from Language View

- Fundamentally, the state part of *Circus* is linked to a B machine and the behavioural part to a CSP specification. Some constructs such as command actions, which specify both state and behaviour, are mapped to constructs in both B and CSP.
- Since in $CSP \parallel B$, the only intersection between a CSP program and a B machine is through the parallel of operations in B with operational events in CSP. Therefore, our link transforms all interactions in *Circus* between the state part and the behavioural part to schema expressions as action, which are finally linked to operations in B and operational events in CSP.

And from the structural view, a *Circus* specification is composed of several sections: global definitions, channel declarations, a set of explicitly defined processes, and compound processes. And in $CSP \parallel B$, a B machine has a definition part, a state part and an operation part, while a CSP specification consists of definitions, channel declarations, and processes. The overall strategies of Υ from the structural view are given below.

- The definitions, such as type definitions, abbreviation and axiomatic definitions, are mapped to the counterparts in B and possibly in CSP if they are referred in the behavioural part of *Circus*.
- Channel declarations are mapped to channel declarations in CSP.
- State components in *Circus* are mapped to variables in B. However, since state components are encapsulated in explicitly defined processes, they are merged to form variables in B when mapped.
- Operational schemas within an explicitly defined process, which includes the state schema in its declaration and is not included by other schemas, are mapped to operations in B. However, these operations are restricted to manipulate variables that

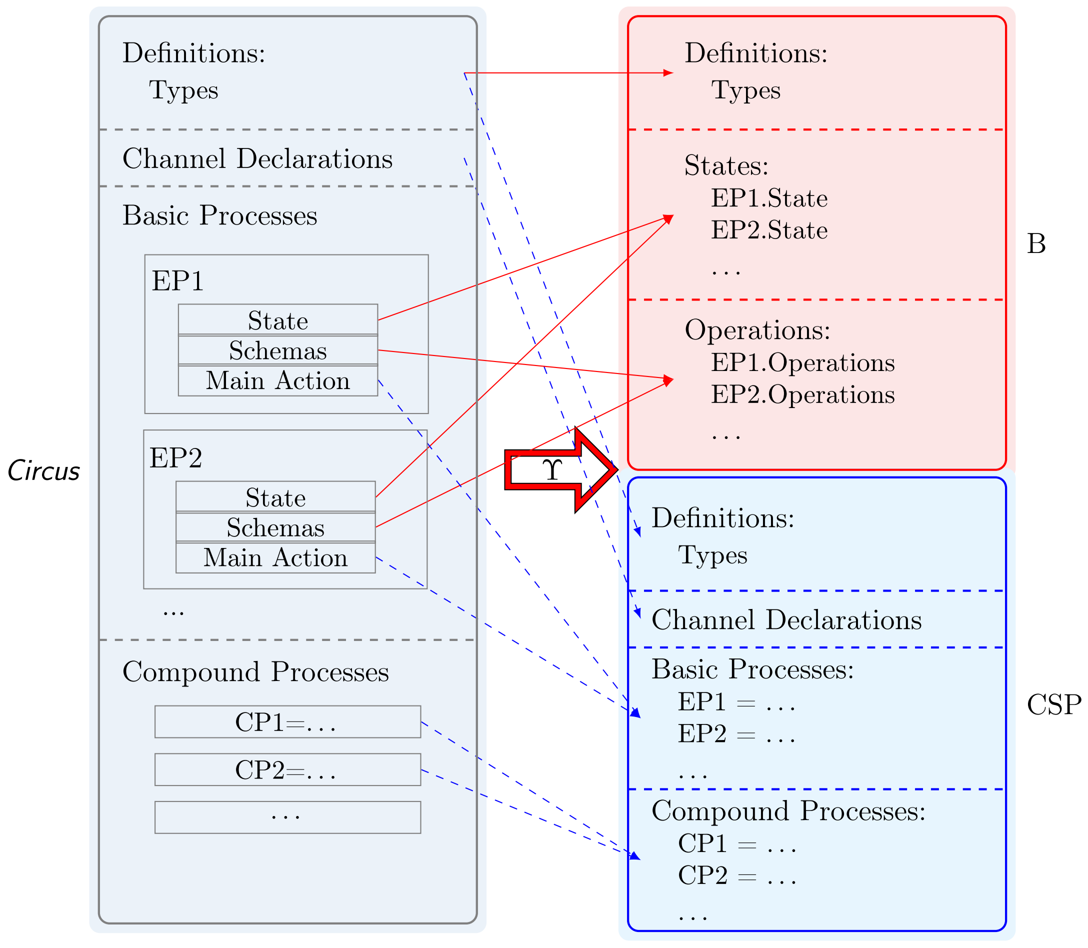

Figure 4.2: Link Relation from Structural View

are mapped from state components of the same process in *Circus*, and never change variables that are mapped from state components of different processes.

- The main action of an explicitly defined process is mapped to a same name process in CSP.
- Compound processes are linked to the same name processes in CSP.

4.1.2 Υ Function Decomposition

Because a *Circus* program is linked to a $CSP \parallel B$ program with a complete B machine and a CSP specification, we decompose the Υ function into two functions: Ω (pronounced "omega") function and Φ (pronounced "phi") function. The Ω function is responsible for the translation of the state part in *Circus* to B, while the Φ function is for that of the behavioural part to CSP.

However, *Circus* itself is not a simple combination of CSP and Z languages but a free mixture of CSP and Z with additional guarded commands. An example is the assignment command that may specify both state and behaviour. For instance, the action

$$c?x \rightarrow (s := x + l)$$

inputs a value x over the channel c, then the assignment command updates the state variable s to x plus the local variable l. In this action, state and behaviour are mixed together. The state part states that s will be updated and after that it is equal to the addition of x and l, while the behavioural part specifies that this update only happens after receiving message on the channel c. As a result, Ω and Φ functions cannot apply to the original *Circus* program directly.

Thus, another function, named R_{wrt}, is defined. It aims to rewrite a *Circus* program to separate the state and behavioural parts into Z and CSP. The action above is rewritten to an action

$$c?x \rightarrow \big(AssOp\big)$$

and a schema

$$AssOp == [\, \Delta P\_StPar \,;\, l? : T_{l?} \,;\, x? : T_c \mid s' = x? + l? \land u' = u \,]$$

according to R_{wrt} Rule 39 which is defined later. Finally, Ω and Φ can be easily applied to this rewritten *Circus* program.

The relation of Υ function decomposition is displayed in Figure 4.3. In a rewritten *Circus* program, state and behaviour are separate. No construct will specify both state and behaviour at the same time. The interaction between them highly depends on schema expressions. The original schemas and schema expressions in Z and behaviour respectively are still kept in the rewritten program. In addition, it is worth noting that additional operational schemas are added in Z, and any direct access and update of state components in *Circus* actions will rely on schema expressions. For instance, provided s is a state component in the action $c!s \rightarrow \mathbf{Skip}$, according to R_{wrt} Rule 27, an extra schema Op_s (where $Op\_s == [\, \Xi StPar ; s! : Ts \mid s! = s \,]$) is added in the Z part and a schema expression $\big(OP\_s\big)$ is added in the beginning of this action to get the value of s in the behavioural part. Furthermore, for other constructs such as commands, they are rewritten to additional schemas and their schema expressions as well. Finally we state that the rewritten *Circus* program has the same structure as the original program, which means state components of each process are still encapsulated in its own process.

4.1.2.1 Ω Function Decomposition

For the Ω function, our strategy is to reuse the currently available solution [85] in ProB to translate Z in ZRM [4] , or ZRM Z for short, to B. Considering this strategy, we map the state part of a *Circus* program to ISO Standard Z first because *Circus* itself is written in ISO Standard Z, then to Z in ZRM, and finally from ZRM to B by ProB. Accordingly, the Ω function is decomposed as well: the Ω_1 function translates the state part in Z in a rewritten *Circus* specification to a complete specification in ISO Standard Z by merging all state components and schemas from all processes; the Ω_2 function syntactically transforms Z in ISO Standard Z to that in ZRM; the Ω_3 function, translation function from ZRM to B, is implemented in ProB and stated in Daniel Plagge et al.'s work [85]. Therefore, the link rules or definitions given in the rest of this paper mainly focus on the link from the original *Circus* specification to a combination of CSP and Z in ZRM. And the link from Z to B is done by ProB and will not be given in this thesis. Finally, an additional notation $\|_B$ is introduced to the combination of CSP and Z in ZRM, and $CSP \|_B Z$ denotes the combination of CSP and Z in ZRM in terms of $CSP \parallel B$ and it is eventually translated by ProB to $CSP \parallel B$.

4.1.3 Link Strategies

In addition to the overall strategies and link functions, some other strategies are defined.

- Every rule defined for Υ is sound unless stated otherwise. The soundness of the map is based on UTP semantics. If the corresponding linked constructs in $CSP \parallel B$ have the same semantics as the original constructs in *Circus*, then the link is sound.

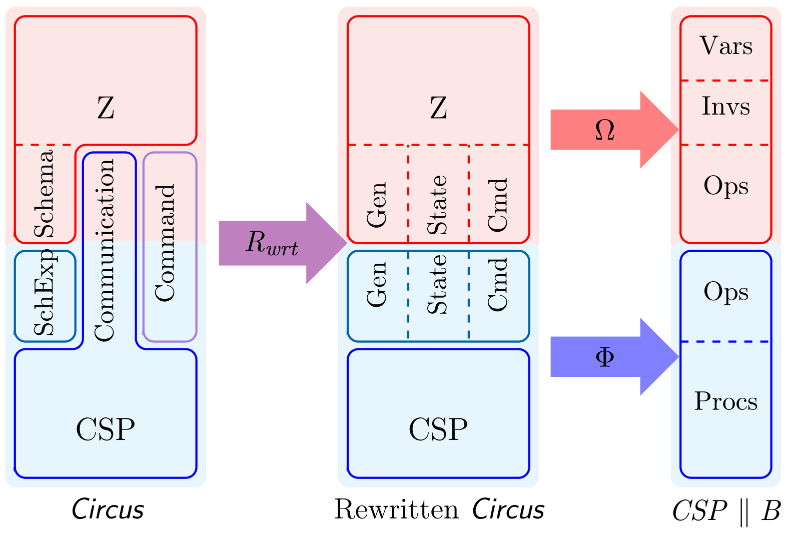

Figure 4.3: Translation Function (Υ) Decomposition

  – From the design perspective, a design $P_1 \vdash Q_1$ is equal to another design $P_2 \vdash Q_2$ if, and only if, $(P_1 = P_2) \wedge (P_1 \Rightarrow (Q_1 = Q_2))$. If both designs have the same alphabets (ok, v, and their dashed counterparts), the same preconditions that imply the equal postcondition, we say they are semantically equal.
  – From the reactive process ($\mathbf{R}(P \vdash Q)$) perspective, if both reactive processes have the same alphabets (ok, $wait$, tr, ref, v, and their dashed counterparts), the same preconditions that imply the equal postcondition and the same other observation variables, we say two reactive processes are semantically equal.
  – For state-based specification languages such as Z and B, their semantics are specified in the design theory of UTP. But for CSP and the behavioural part of *Circus*, their semantics are specified in the reactive theory of UTP.

- State components of *Circus* are maintained in a Z specification and finally a B machine. Thus we require they are updated only in the B machine but can be accessed in both B and CSP programs. The CSP specification will not maintain states. If a process in the CSP specification needs to get the value of variables in B, it shall retrieve them through a communication between CSP and B.

4.2 *Circus* Common Constructs Translation

4.2.1 Identifiers

In ISO Z Standard, an identifier is a decorated word, DECORWORD, composed of WORD and STROKE. A WORD can be a keyword, an operator or a NAME.

A STROKE (`'`, `!`, and `?`) can be used in three contexts [70, 8.4].

- NAME, such as q'.
- binding construction expression if it can be interpreted, such as $\theta\, S'$ provided S is a schema.
- schema decoration expression, such as S' provided S is a schema.

In addition to LETTER, DIGITAL and underscore (_,_), a NAME may have other special symbols, such as

- stroke (`'`, `!`, and `?`)
- subscript x_1 (`x_1`)
- superscript x^1 (`x^1`)
- ...

4.2.1.1 Identifiers in CSP$_M$ and B

The pattern of an identifier or name in CSP$_M$ and B: `[a-zA-Z][a-zA-Z0-9_]` [60] [86]. It begins with an alphabetic character (`[a-zA-Z]`) and are followed by any number of alphanumeric characters or underscores. Particularly, for CSP$_M$, it can be optionally followed by a prime character (`'`).

4.2.1.2 Consideration

Because a name in CSP$_M$ and B has limited patterns but in *Circus* it has more, our translation needs to cope with name translation as well. Two potential solutions are immediately presented.

- Replace special symbols with specific combination of characters in CSP$_M$ and B. For example, x_1 may be translated to `x_1` to map a subscript to an underscore;
- Restrict symbols used in *Circus* to make translation easy. For example, x_1 is not allowed.

We use the second solution because for the first solution we still need to cope with naming conflict—x_1 may have been used before translating a x_1 to `x_1`. Furthermore, usually the common naming pattern is frequently used when writing *Circus* program, such as x_1 instead of x_1.

However, some special symbols like stroke (`'`, `!`, and `?`) in a name are very important parts of Z: x' within a schema or a specification statement means after-state of the variable x; $x?$ and $x!$ within a schema mean input and output variables of the schema. Thus these symbols have to be kept. By our rules, they are translated to ZRM and B finally. In B machine, the after-state of a variable is specified by substitution and occurs in RHS of substitution, such as `x := x + 1`. Therefore, prime ($'$, `'`) in x' will be removed. For input and output variables, a B operation, (`o1, o2, ... <-- Op(i1, i2, ...) = ...`), has specific position for input and output variables in the definition of an operation. Thus ? and ! strokes are removed as well.

In sum, our rule is to restrict the name pattern in *Circus* to have the pattern below.

```
[a-zA-Z][a-zA-Z0-9_]['!?]
```

The end prime strokes are only used in the schema and specification statement contexts while input stroke and output stroke are only used in the schema context.

4.3 *Circus* Rewriting Function - R_{wrt}

The R_{wrt} function aims for the separation of the state part and the behavioural part in explicitly defined processes. Consequently, all interactions between the state part and the behavioural part are through schema expressions only.

Among a large number of rewrite rules, most of them are straightforward while only some of them, listed as follows, are not.

- Inheriting Sections: R_{wrt} Rule 2

- Axiomatic Definitions: R_{wrt} Rule 5
- Parametrised Processes: R_{wrt} Rule 11
- Indexed Processes: R_{wrt} Rule 14
- Renaming Operator: R_{wrt} Rule 19
- Explicitly Defined Processes with Renaming: R_{wrt} Rule 20
- Indexed Processes with Renaming: R_{wrt} Rule 21
- External choice of processes in R_{wrt} Rule 22
- Additional State Components Retrieve Schemas: R_{wrt} Rule 23
- Prefixing Action: R_{wrt} Rule 27
- Guarded Action: R_{wrt} Rule 28
- External Choice: R_{wrt} Rule 30
- Recursion: R_{wrt} Rule 34
- Assignment: R_{wrt} Rule 39
- Alternation: R_{wrt} Rule 40
- Parametrisation by Value, by Result, and by Value-Result: R_{wrt} Rule 42
- Specification Statement: R_{wrt} Rule 43

4.3.1 General Rule

R_{wrt} **Rule 1 (General Rule).** For all constructs of which rewrite rules are not specifically defined in this section, their rewrites will not change anything.

4.3.2 Inheriting Sections

In addition to the syntax of *Circus* briefed in Section 2.2.1, a *Circus* program can be organized and reused in the same mechanism as in ISO Standard Z by inheriting sections. For example, the section header below has standard *circus_toolkit* as its parent and it inherits and reuses all definitions in *circus_toolkit* easily.

$\mathbf{section}\ a\ \mathbf{parents}\ circus\_toolkit$

And another example is shown below. If a section b has a as its parent, then b can reuse all definitions in a including all from *circus_toolkit*.

$\mathbf{section}\ b\ \mathbf{parents}\ a$

R_{wrt} **Rule 2 (Inheriting Sections).** For a section with n parent sections,

$\mathbf{section}\ this\_section\ \mathbf{parents}\ psec_1, psec_2, \ldots, psec_n$

it is rewritten to a new section with the same section name but all definitions from its parent sections recursively included except standard toolkits. Finally, it results in the new section below.

$\mathbf{section}\ this\_section\ \mathbf{parents}\ standard\_toolkit, \ldots$

$$\begin{array}{l} AllDef(psec_1) \\ AllDef(psec_2) \\ \vdots \\ AllDef(psec_n) \end{array}$$

where $AllDef(sec)$ denotes all definitions from the section sec, and all its parent sections only include standard toolkits.

The definition of $AllDef$ is given below.

Step 1 If sec only has standard toolkits as its parent sections, then $AllDef(sec)$ returns all definitions in this section. Otherwise,

Step 2 If sec has other parent sections, it includes all definitions from these sections by $AllDef$ according to the order of these sections' name in its section header. In addition, all definitions from parent sections are put in the front of this section.

Step 3 For $AllDef$ of its parent sections, repeat "Step 1" and "Step 2" above to include all definitions.

Step 4 If its parent sections have the same sections as their parent sections, include only one copy of these grandparent sections before these parent sections.

□

4.3.3 Expressions, Predicates and Operators

R_{wrt} **Rule 3 (Expressions, Predicates and Operators).** Rewriting an expression, a predicate or an operator by R_{wrt} results in the same construct. □

4.3.4 Given Set Definitions

Given sets, or basic types, are a way to introduce types into the specification. For a given set, only its name matters and its internal structure is of no interest—these elements cannot be identified and compared. However, there is no corresponding set notation in CSP. Therefore, given sets are rewritten.

R_{wrt} **Rule 4 (Given Sets).** For each given set defined below,

$$[GSet_1, \ldots, GSet_n]$$

it is rewritten to a free type

$$GSet_i ::= CGSet_i \langle\langle 1 \mathrel{..} MAX\_GSET\_INTS \rangle\rangle$$

where MAX_GSET_INTS is a constant introduced to denote maximum elements in the given set.

4.3.5 Axiomatic Definitions

An axiomatic definition introduces a set of global variables with a constraint on them. When these constants are linked to constants in CSP finally, they must be instantiated because constants in CSP are concrete. In addition, the instances in CSP must match their values in Z. Thus in this stage, axiomatic definitions in *Circus* are rewritten by appending additional constraints into the predicate of the definition to choose only one instance of the constants. Then the rewritten axiomatic definitions are mapped to the same axiomatics in Z by the Ω function and constants in CSP by Φ Rule 3. The consistency of constants in Z and CSP is therefore preserved.

R_{wrt} **Rule 5 (Axiomatic Definitions).** If the global constants defined in an axiomatic definition are used in the behavioural part—which finally is linked to CSP, additional constraints are added to its predicate to choose only one instance of all defined constants. For example, the axiomatic definition

$$\begin{array}{|l}
size, max\_size : \mathbb{N} \\
\hline
size < max\_size
\end{array}$$

is rewritten to

$$\begin{array}{|l}
size, max\_size : \mathbb{N} \\
\hline
size < max\_size \\
size = 5 \\
max\_size = 15
\end{array}$$

where 5 and 15 for *size* and *max_size* are just one valid instance. $\square$

4.3.6 Channel Declarations

R_{wrt} **Rule 6 (Channel Declarations).** A schema grouped channel declaration is expanded to one or more typed channel declarations that depend on the number of variable declarations in the schema.

$$\begin{array}{lcl}
R_{wrt}\,(\textbf{channel}\ c_1, \cdots, c_n) & = & \textbf{channel}\, c_1, \cdots, c_n \\
R_{wrt}\,(\textbf{channel}\ c_1, \cdots, c_n : T) & = & \textbf{channel}\, c_1, \cdots, c_n : T \\
R_{wrt}\,(\textbf{channelfrom}\ S) & = & \left\{ \begin{array}{l} \textbf{channel}\ c_1, c_2, \ldots, c_n : T_c \\ \textbf{channel}\ d_1, d_2, \ldots, d_m : T_d \end{array} \right.
\end{array}$$

provided S is a schema defined below and *pred* is omitted because only restriction imposed on communication is its type.

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\
\hline
c_1, c_2, \ldots, c_n : T_c \\
d_1, d_2, \ldots, d_n : T_d \\
\hline
pred \\
\hline
\end{array}$$

$\square$

4.3.7 Channel Set Declarations

R_{wrt} **Rule 7 (Channel Set Declarations).** Rewrite of channel set declarations is the rewrite of their channel set expressions.

$$R_{wrt}\,(\textbf{channelset}\, N == CSExp) \quad = \quad \textbf{channelset}\, N == R_{wrt}\,(CSExp)$$

$\square$

4.3.8 Channel Set Expressions

R_{wrt} **Rule 8 (Channel Set Expressions).** Rewrite of channel set expressions remains the same.

$$\begin{array}{lcl}
R_{wrt}\,(\{\!|\ |\!\}) & = & \{\!|\ |\!\} \\
R_{wrt}\,(\{\!|\, c_1, c_2, \cdots, c_n \,|\!\}) & = & \{\!|\, c_1, c_2, \cdots, c_n \,|\!\} \\
R_{wrt}\,(CSExp) & = & CSExp
\end{array}$$

$\square$

4.3.9 Name Set Declarations

R_{wrt} **Rule 9 (Name Set Declarations).** Rewrite of name set declarations is the rewrite of their name set expressions.

$$R_{wrt}\,(\mathbf{nameset}\,n == \mathit{NSExp}) \quad = \quad \mathbf{nameset}\,n == R_{wrt}\,(\mathit{NSExp})$$

□

4.3.10 Name Set Expressions

R_{wrt} **Rule 10 (Name Set Expressions).** Rewrite of name set expressions remains the same.

$$\begin{array}{lcl} R_{wrt}\,(\{\,\}) & = & \{\,\} \\ R_{wrt}\,(\{\,x\,\}) & = & \{\,x\,\} \\ R_{wrt}\,(\mathit{NSExp}) & = & \mathit{NSExp} \end{array}$$

□

4.3.11 Parametrised Processes

R_{wrt} **Rule 11 (Parametrised Processes).** For a parametrised process, it is expanded to a number of explicitly defined processes. Provided that a parametrised process with m parameters, named $x_1, x_2, \cdots, x_m$, and they have corresponding types $T_1, T_2, \cdots, T_m$. And for each type T_i, it has n_i elements. Finally, the notion x_{ij} is used to represent the jth element of the ith parameter. The number of explicitly defined processes is equal to the multiplication of the cardinality of all types: $\mathit{card}(T_1) \times \mathit{card}(T_2) \times \cdots \times \mathit{card}(T_m)$. For each combination of parameters $(x_1, x_2, \cdots, x_n)$, it is mapped to an explicitly defined process whose name is given by the *strcat* function.

$$R_{wrt}\,(\mathbf{process}\ \ PP \mathrel{\widehat=} x_1 : T_1\,;\, x_2 : T_2\,;\, \cdots\,;\, x_m : T_m \bullet P)$$

$$= \left(\begin{array}{l} R_{wrt}\left(\begin{array}{c} \mathbf{process}\ \ \mathit{strcat}\,(PP, \_, [x_{11}, x_{21}, \cdots, x_{m1}]) \\ \mathrel{\widehat=} P[x_{11}/x_1, x_{21}/x_2, \cdots, x_{m1}/x_m] \end{array}\right) \\ R_{wrt}\left(\begin{array}{c} \mathbf{process}\ \ \mathit{strcat}(PP, \_, [x_{11}, x_{21}, \cdots, x_{m2}]) \\ \mathrel{\widehat=} P[x_{11}/x_1, x_{21}/x_2, \cdots, x_{m2}/x_m] \end{array}\right) \\ \vdots \\ R_{wrt}\left(\begin{array}{c} \mathbf{process}\ \ \mathit{strcat}(PP, \_, [x_{11}, x_{21}, \cdots, x_{mn_m}]) \\ \mathrel{\widehat=} P[x_{11}/x_1, x_{21}/x_2, \cdots, x_{mn_m}/x_m] \end{array}\right) \\ \vdots \\ R_{wrt}\left(\begin{array}{c} \mathbf{process}\ \ \mathit{strcat}(PP, \_, [x_{1n_1}, x_{2n_2}, \cdots, x_{mn_m}]) \\ \mathrel{\widehat=} P[x_{1n_1}/x_1, x_{2n_2}/x_2, \cdots, x_{mn_m}/x_m] \end{array}\right) \end{array}\right)$$

where

- P is an explicitly defined process definition (**begin** $\cdots$ **end**). If P is a reference to an explicitly defined process, then P in the rule should be replaced by the body of P: $B(P)$ where B is a function to get the body of the process P
- the substitution notation $P[x_1/x]$ denotes the expression x_1 consistently substituted for free occurrences of the variable x in P
- *strcat* is a string concatenation function defined in Definition B.2.16

For a parametrised process with only one formal parameter,

$$R_{wrt}\,(\textbf{process}\ \ PP \mathrel{\widehat=} x : T \bullet P) = \left(\begin{array}{l} R_{wrt}\,(\textbf{process}\ \ PP\_x_1 \mathrel{\widehat=} P[x_1/x]) \\ \vdots \\ R_{wrt}\,(\textbf{process}\ \ PP\_x_n \mathrel{\widehat=} P[x_n/x]) \end{array}\right)$$

□

R_{wrt} **Rule 12 (Parametrised Process Invocation).** The rewrite of a parametrised process invocation is to find the corresponding mapped basic process

$$\begin{array}{l} R_{wrt}\,(PP(e_1, e_2, \cdots, e_m)) \\ = \left(\begin{array}{ll} & (e_1 = x_{11} \wedge e_2 = x_{21} \wedge \cdots \wedge e_m = x_{m1}) \\ & \qquad \& strcat\,(PP, \_, [x_{11}, x_{21}, \cdots, x_{m1}]) \\ \Box & (e_1 = x_{11} \wedge e_2 = x_{21} \wedge \cdots \wedge e_m = x_{m2}) \\ & \qquad \& strcat\,(PP, \_, [x_{11}, x_{21}, \cdots, x_{m2}]) \\ \vdots & \\ \Box & (e_1 = x_{11} \wedge e_2 = x_{21} \wedge \cdots \wedge e_m = x_{mn_m}) \\ & \qquad \& strcat\,(PP, \_, [x_{11}, x_{21}, \cdots, x_{mn_m}]) \\ \vdots & \\ \Box & (e_1 = x_{1n_1} \wedge e_2 = x_{2n_2} \wedge \cdots \wedge e_m = x_{mn_m}) \\ & \qquad \& strcat\,(PP, \_, [x_{1n_1}, x_{2n_2}, \cdots, x_{mn_m}]) \end{array}\right) \end{array}$$

This resultant guarded processes are not valid syntax in *Circus*. They are just intermediate representations and will be linked to CSP in the later stage. Finally their corresponding constructs in CSP are valid.

If the parametrised process only has one formal parameter, then its invocation is simplified.

$$R_{wrt}\,(PP(e)) = \left(\begin{array}{ll} & (e = x_1)\ \ \& \ PP\_x_1 \\ \Box & (e = x_2)\ \& PP\_x_2 \\ & \vdots \\ \Box & (e = x_n)\ \& PP\_x_n \end{array}\right)$$

□

R_{wrt} **Rule 13 (Anonymous Parametrised Process and its Invocation).** The invocation of the parametrised process can be an anonymous parametrised process. It is rewritten to a named parametrised process and its invocation with the same actual parameters.

$$R_{wrt}\,((x : T \bullet P)\,(e)) = \left\{\begin{array}{l} R_{wrt}\,(\textbf{process}\ UPP \mathrel{\widehat=} x : T \bullet P) \\ R_{wrt}\,(UPP(e)) \end{array}\right.$$

After that, they are rewritten by R_{wrt} Rule 11 and R_{wrt} Rule 12. □

4.3.12 Indexed Processes

R_{wrt} **Rule 14 (Indexed Processes).** An indexed process is rewritten to a parametrised process with all its channels renamed at first, then it is expanded to a number of explicitly defined processes by the parametrised process rule.

$$\begin{array}{ll} R_{wrt}\,(\textbf{process}\ \ IP \mathrel{\widehat=} i : T \odot P) & \\ = R_{wrt}\,(\textbf{process}\ \ IP \mathrel{\widehat=} i : T \bullet P[c := c\_i.i]) & \text{[Definition of indexed processes]} \end{array}$$

$$= \left(\begin{array}{l} R_{wrt}\left(\textbf{process}\ \ IP\_i_1 \mathrel{\widehat=} (P[c := c\_i.i])[i_1/i]\right) \\ \vdots \\ R_{wrt}\left(\textbf{process}\ \ IP\_i_n \mathrel{\widehat=} (P[c := c\_i.i])[i_n/i]\right) \end{array} \right) \qquad [R_{wrt}\text{ Rule 11}]$$

$$= \left(\begin{array}{l} R_{wrt}\left(\textbf{process}\ \ IP\_i_1 \mathrel{\widehat=} P[c := c\_i.i_1]\right) \\ \vdots \\ R_{wrt}\left(\textbf{process}\ \ IP\_i_n \mathrel{\widehat=} P[c := c\_i.i_n]\right) \end{array} \right)$$

[Substitution and i does not occur in P]

where

- P is an explicitly defined process definition (**begin**$\cdots$**end**). If P is a reference to an explicitly defined process, then P in the rule should be replaced by the body of P: $B(P)$
- $P[c := c\_i.i]$ denotes the renaming of each channel c in P to $c\_i.i$, and $T = \{i_1, \cdots, i_n\}$.

$\square$

R_{wrt} **Rule 15 (Indexed Process Invoication).**

$$R_{wrt}\left(IP\lfloor e \rfloor\right) = \left(\begin{array}{ll} & (e = i_1)\ \&\ IP\_i_1 \\ \square & (e = i_2)\ \&IP\_i_2 \\ & \vdots \\ \square & (e = i_n)\ \&IP\_i_n \end{array} \right)$$

$\square$

R_{wrt} **Rule 16 (Anonymous Indexed Process and its Invocation).** The invocation of an indexed process can be an anonymous indexed process. It is rewritten to a named indexed process and its invocation with the same actual parameters.

$$R_{wrt}\left((x : T \odot P)\lfloor e \rfloor\right) = \left\{ \begin{array}{l} R_{wrt}\left(\textbf{process}\ \ UIP \mathrel{\widehat=} x : T \odot P\right) \\ R_{wrt}\left(UIP\lfloor e \rfloor\right) \end{array} \right.$$

After that, they are rewritten be the R_{wrt} Rule 14 and R_{wrt} Rule 15. $\square$

4.3.13 Process Definition

R_{wrt} **Rule 17 (Process Definition).** The rewrite of a process definition except the parametrised process, the indexed process and the renamed process, is a same name process with its body rewritten.

$$R_{wrt}\left(\textbf{process}\, P \mathrel{\widehat=} B(P)\right) = \textbf{process}\, P \mathrel{\widehat=} R_{wrt}\left(B(P)\right)$$

$\square$

4.3.14 Process Invocation

R_{wrt} **Rule 18 (Process Invocation).** The rewrite of a process invocation except the parametrised process and the indexed process is the name of the process.

$$R_{wrt}\left(P\right) = P$$

$\square$

4.3.15 Renaming Operator

The renaming operator $P[c_{old} := c_{new}]$ defines a new process like P, except that all channels in c_{old} of P are renamed to the corresponding channels in c_{new}. It is worth noting that all channels in c_{new} are implicitly declared, if needed.

R_{wrt} **Rule 19 (Renaming Operator).** In this rule, only process definitions by the renaming operator are taken into account and processes that are defined having the renaming operator in the middle of their definitions are excluded. Therefore, this rule is eligible for the process RP

$$\textbf{process}\, RP \mathrel{\widehat=} P[c_{old} := c_{new}]$$

but not eligible for the process PQ below because the renaming operator is used in the middle of the definition of PQ.

$$\textbf{process}\, PQ \mathrel{\widehat=} (P[c_{old} := c_{new}]) \mathbin{;} Q$$

And additionally, provided P is only an explicitly defined process or an indexed process, then

$$\begin{array}{l} R_{wrt}\,(\textbf{process}\, RP \mathrel{\widehat=} P[c_{old} := c_{new}]) \\ \quad = R_{wrt}\,(\textbf{process}\, RP \mathrel{\widehat=} F_{Ren}\,(P, \{(c_{old}, c_{new})\})) \end{array}$$

where

- P is an explicitly defined process definition ($\textbf{begin} \cdots \textbf{end}$). If P is a reference to an explicitly defined process, then P in the rule should be replaced by the body of P: $B(P)$
- $F_{Ren}(P, \{(x, y)\})$, defined in Definition B.2.14, is a renaming function that replaces occurrences of the term x in P to the term y

If P is a reference to an explicitly defined process, then P in the rule above should be replaced by $B(P)$. Therefore,

$$\begin{array}{l} R_{wrt}\,(\textbf{process}\, RP \mathrel{\widehat=} P[c_{old} := c_{new}]) \\ \quad = R_{wrt}\,(\textbf{process}\, RP \mathrel{\widehat=} F_{Ren}\,(B(P), \{(c_{old}, c_{new})\})) \end{array}$$

This rule states that the rewrite of a process RP, defined by the renaming of a basic process or an index process P, is equal to the rewrite of a process with P expanded to its body $B(P)$ and all channels in c_{old} are additionally renamed to the corresponding channels in c_{new}. $\square$

R_{wrt} **Rule 20 (Explicitly Defined Processes with Renaming).** If P in R_{wrt} Rule 19 is an explicitly defined process,

$$\begin{array}{l} \textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\ \quad \textbf{state}\, StPar == [\, s_1 : T_{P_1} \,; \cdots s_n : T_{P_n} \mid p \,] \\ \quad Init == [\, (StPar)' \mid pi \,] \\ \quad Pars == [\, \cdots] \\ \quad APars \mathrel{\widehat=} B(APars) \\ \quad \bullet\, A \\ \textbf{end} \end{array}$$

then

$$
\begin{aligned}
& R_{wrt}\,(\mathbf{process}\, RP \mathrel{\widehat=} P[c_{old} := c_{new}]) \\
& = R_{wrt}\left(\begin{array}{l}
\mathbf{process}\, RP \mathrel{\widehat=} \\
F_{Ren}\left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_1 : T_{P_1} ; \cdots s_n : T_{P_n} \mid p \,] \\
\quad Init == [\, (StPar)' \mid pi \,] \\
\quad Pars == [\cdots] \\
\quad APars \mathrel{\widehat=} B(APars) \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}\right), \\
\{(c_{old}, c_{new})\}
\end{array}\right)
\end{array}\right) && [R_{wrt} \text{ Rule 19}] \\
& = R_{wrt}\left(\begin{array}{l}
\mathbf{process}\, RP \mathrel{\widehat=} \\
\left(\begin{array}{l}
\mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_1 : T_{P_1} ; \cdots s_n : T_{P_n} \mid p \,] \\
\quad Init == [\, (StPar)' \mid pi \,] \\
\quad Pars == [\cdots] \\
\quad APars \mathrel{\widehat=} F_{Ren}\,(B(APars), \{(c_{old}, c_{new})\}) \\
\quad \bullet\, F_{Ren}\,(A, \{(c_{old}, c_{new})\}) \\
\mathbf{end}
\end{array}\right)
\end{array}\right) && [\text{Only action definitions and the main action are renamed}] \\
& = \left(\begin{array}{l}
\mathbf{process}\, RP \mathrel{\widehat=} \\
R_{wrt}\left(\begin{array}{l}
\mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_1 : T_{P_1} ; \cdots s_n : T_{P_n} \mid p \,] \\
\quad Init == [\, (StPar)' \mid pi \,] \\
\quad Pars == [\cdots] \\
\quad APars \mathrel{\widehat=} F_{Ren}\,(B(APars), \{(c_{old}, c_{new})\}) \\
\quad \bullet\, F_{Ren}\,(A, \{(c_{old}, c_{new})\}) \\
\mathbf{end}
\end{array}\right)
\end{array}\right) && [R_{wrt} \text{ Rule 17}]
\end{aligned}
$$

Eventually, it is a rewrite of the renamed process body in which the channels, which are in c_{old}, in action definitions and the main action are renamed to the corresponding channels in c_{new}. □

R_{wrt} **Rule 21 (Indexed Processes with Renaming).** In *Circus*, the indexed process notation is commonly used with the renaming operator together to define more expressive processes. Therefore, if the process to be renamed in R_{wrt} Rule 19 is an indexed process IP,

$$\mathbf{process}\;\; IP \mathrel{\widehat=} i : T \odot P$$

where P is a reference to an explicitly defined process. Then

$$
\begin{aligned}
& R_{wrt}\,(\mathbf{process}\, RP \mathrel{\widehat=} IP[c\_i := d]) \\
& = R_{wrt}\,(\mathbf{process}\, RP \mathrel{\widehat=} F_{Ren}\,((i : T \odot P)\,, \{(c\_i, d)\})) && [R_{wrt} \text{ Rule 19}] \\
& = R_{wrt}\,(\mathbf{process}\, RP \mathrel{\widehat=} F_{Ren}\,((i : T \bullet P[c := c\_i.i])\,, \{(c\_i, d)\})) \\
& && [\text{Definition of indexed processes and } R_{wrt} \text{ Rule 14}] \\
& = R_{wrt}\,(\mathbf{process}\, RP \mathrel{\widehat=} i : T \bullet F_{Ren}\,(P[c := c\_i.i], \{(c\_i, d)\})) \\
& && [F_{Ren} \text{ only renames channels here}]
\end{aligned}
$$

$$= R_{wrt}\left(\textbf{process}\, RP \mathrel{\widehat=} i : T \bullet F_{Ren}\left(F_{Ren}\left(B(P), \{(c, c\_i.i)\}\right), \{(c\_i, d)\}\right)\right)$$

[R_{wrt} Rule 19 and P is a reference to an explicitly defined process]

$$= R_{wrt}\left(\textbf{process}\, RP \mathrel{\widehat=} i : T \bullet F_{Ren}\left(B(P), \{(c, d.i)\}\right)\right)$$

[F_{Ren} Transitivity Lemma C.2.2]

$$= \begin{pmatrix} R_{wrt}\left(\textbf{process}\ \ RP\_i_1 \mathrel{\widehat=} F_{Ren}\left(B(P), \{(c, d.i)\}\right)[i_1/i]\right) \\ \vdots \\ R_{wrt}\left(\textbf{process}\ \ RP\_i_n \mathrel{\widehat=} F_{Ren}\left(B(P), \{(c, d.i)\}\right)[i_n/i]\right) \end{pmatrix} \qquad [R_{wrt} \text{ Rule } 11]$$

$$= \begin{pmatrix} R_{wrt}\left(\textbf{process}\ \ RP\_i_1 \mathrel{\widehat=} F_{Ren}\left(B(P), \{(c, d.i_1)\}\right)\right) \\ \vdots \\ R_{wrt}\left(\textbf{process}\ \ RP\_i_n \mathrel{\widehat=} F_{Ren}\left(B(P), \{(c, d.i_n)\}\right)\right) \end{pmatrix}$$

[Definition of indexed processes that the parameter i does not occur in P]

$$= \begin{pmatrix} \textbf{process}\ \ RP\_i_1 \mathrel{\widehat=} R_{wrt}\left(F_{Ren}\left(B(P), \{(c, d.i_1)\}\right)\right) \\ \vdots \\ \textbf{process}\ \ RP\_i_n \mathrel{\widehat=} R_{wrt}\left(F_{Ren}\left(B(P), \{(c, d.i_n)\}\right)\right) \end{pmatrix} \qquad [R_{wrt} \text{ Rule } 17]$$

Finally, it results in a set of explicitly defined processes like the rewrite of an indexed process, but for each channel c in P, it is renamed to $d.i_j$ (where j is an index within $1 \mathinner{..} n$) for each individual explicitly defined process RP_i_j. □

4.3.16 Compound Processes

R_{wrt} **Rule 22 (Compound Processes).** For compound processes defined in term of CSP operators, their rewrites are straightforward. The only exception is external choice of two processes, where PA, a prefixed process given in Definition B.4.1, is like the process P but its main action is a prefixed action AA (Definition B.3.1).

$$\begin{array}{ll}
R_{wrt}\left(P_1 \mathbin{;} P_2\right) & = R_{wrt}\left(P_1\right) \mathbin{;} R_{wrt}\left(P_2\right) \\
R_{wrt}\left(PA_1 \mathbin{\Box} PA_2\right) & = R_{wrt}\left(PA_1\right) \mathbin{\Box} R_{wrt}\left(PA_2\right) \\
R_{wrt}\left(P_1 \sqcap P_2\right) & = R_{wrt}\left(P_1\right) \sqcap R_{wrt}\left(P_2\right) \\
R_{wrt}\left(P_1 \llbracket\, cs \,\rrbracket\, P_2\right) & = R_{wrt}\left(P_1\right) \llbracket\, cs \,\rrbracket\, R_{wrt}\left(P_2\right) \\
R_{wrt}\left(P_1 \mathbin{|||} P_2\right) & = R_{wrt}\left(P_1\right) \mathbin{|||} R_{wrt}\left(P_2\right) \\
R_{wrt}\left(P \setminus cs\right) & = R_{wrt}\left(P\right) \setminus cs \\
R_{wrt}\left(\mathop{;} x : T \bullet P(x)\right) & = \mathop{;} x : T \bullet R_{wrt}\left(P(x)\right) \\
R_{wrt}\left(\Box\, x : T \bullet PA(x)\right) & = \Box\, x : T \bullet R_{wrt}\left(PA(x)\right) \\
R_{wrt}\left(\bigsqcap x : T \bullet P(x)\right) & = \bigsqcap x : T \bullet R_{wrt}\left(P(x)\right) \\
R_{wrt}\left(\llbracket\, CS \,\rrbracket\, x : T \bullet P(x)\right) & = \llbracket\, CS \,\rrbracket\, x : T \bullet R_{wrt}\left(P(x)\right) \\
R_{wrt}\left(\big|\big|\big|\, x : T \bullet P(x)\right) & = \big|\big|\big|\, x : T \bullet R_{wrt}\left(P(x)\right)
\end{array}$$

□

4.3.17 Explicitly Defined Processes

The rewrite of explicitly defined processes is to separate the state part and the behavioural part as well as rename state components, schema paragraphs and action paragraphs. Consequently, all interactions between the state part and the behavioural part are through schema expressions only.

Basically, the rewrite of basic processes consists of several steps.

- Firstly, state components retrieve schemas are added in the process for actions to get their values through schema expressions as action.

- Then, all state components, schema paragraphs, and action paragraphs in the process are renamed to avoid name conflicts when this process is merged with other processes in the later stage by Ω_1 Rule 1.

- Finally, the main action is rewritten.

4.3.17.1 Additional State Components Retrieve Schemas

R_{wrt} **Rule 23 (Additional State Components Retrieve Schemas).** The rule for state components retrieve schemas is shown below, where the B function denotes the body of the action. For each state component s_i in an explicitly defined process, one schema OP_s_i is added to retrieve the value of this state component. The name of the output variable in this schema is composed of the state component name and !. And its type is the same as the type of the state component.

$$
\begin{array}{l}
R_{wrt}\left(\begin{array}{l}
\mathbf{process}\ \ P \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, \mathit{StPar} == [\, s_1 : T_1\, ; \cdots s_n : T_n \mid p\,] \\
\quad \mathit{Pars} == [\cdots] \\
\quad \mathit{APars} \mathrel{\widehat=} B(\mathit{APars}) \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}\right) \\
= \left(\begin{array}{l}
\mathbf{process}\ \ P \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, \mathit{StPar} == [\, s_1 : T_1\, ; \cdots s_n : T_n \mid p\,] \\
\quad \mathit{Pars} == [\cdots] \\
\quad \mathit{Op\_s_1} == [\, \Xi \mathit{StPar}\, ; s_1! : T_1 \mid s_1! = s_1\,] \\
\quad \vdots \\
\quad \mathit{Op\_s_n} == [\, \Xi \mathit{StPar}\, ; s_n! : T_n \mid s_n! = s_n\,] \\
\quad \mathit{APars} \mathrel{\widehat=} B(\mathit{APars}) \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}\right)
\end{array}
$$

□

4.3.17.2 Renaming

R_{wrt} **Rule 24 (Renaming of State Components, Schemas, Actions and their Refereneces).** State components, schema paragraphs, action paragraphs, and each reference to them within an explicitly defined process are renamed by prefixing the process's name. The only exception is that the reference to state components in action is not changed.

$$R_{wrt}\left(\begin{array}{l}\textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\, StPar == [\, s_1 : T_1 \,;\cdots s_n : T_n \mid p \,]\\ \quad Pars == [\cdots]\\ \quad Op\_s_1 == [\,\Xi StPar \,;\, s_1! : T_1 \mid s_1! = s_1\,]\\ \quad \cdots\\ \quad Op\_s_n == [\,\Xi StPar \,;\, s_n! : T_n \mid s_n! = s_n\,]\\ \quad APars \mathrel{\widehat=} B(APars)\\ \quad \bullet\ A\\ \textbf{end}\end{array}\right)$$

$$= \left(\begin{array}{l}\textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\, P\_StPar == [\, P\_s_1 : T_1 \,;\cdots P\_s_n : T_n \mid p \,]\\ \quad P\_Pars == [\cdots]\\ \quad P\_Op\_s_1 == [\,\Xi P\_StPar \,;\, s_1! : T_1 \mid s_1! = P\_s_1\,]\\ \quad \cdots\\ \quad P\_Op\_s_n == [\,\Xi P\_StPar \,;\, s_n! : T_n \mid s_n! = P\_s_n\,]\\ \quad P\_APars \mathrel{\widehat=} B(P\_APars)\\ \quad \bullet\ R_{wrt}(A)\\ \textbf{end}\end{array}\right)$$

□

4.3.17.3 Action Rewriting

R_{pre} **and** R_{post}

Definition 4.3.1 (R_{pre} and R_{post}). *Rewriting an action to get the value of state components in its first construct, $R_{wrt}(A)$, is composed of $R_{pre}(A)$ and $R_{post}(A)$ which denotes the prefix (state components retrieve schema expressions) and the remaining respectively:*

- $R_{wrt}(A) = R_{pre}(A) \rightarrow R_{post}(A)$, *if $R_{pre}\,(A)$ is not empty*
- $R_{wrt}(A) = R_{post}(A)$, *if $R_{pre}\,(A)$ is empty*

For example,

Constructs	R_{pre}	R_{post}
Skip		**Skip**
Stop		**Stop**
$c!s_i!\ldots!s_j \rightarrow A$	$\big(OP\_s_i\big) \rightarrow \cdots \rightarrow \big(OP\_s_j\big)$	$c!s_i!\ldots!s_j \rightarrow R_{wrt}(A)$
g	$\big(OP\_s_i\big) \rightarrow \cdots \rightarrow \big(OP\_s_j\big)$	g
$\neg g_1$	$R_{pre}(g_1)$	$\neg g_1$
$g_1 \wedge g_2$	$R_{pre}(g_1) \rightarrow R_{pre}(g_2)$	$g_1 \wedge g_2$
$g_1 \vee g_2$	$R_{pre}(g_1) \rightarrow R_{pre}(g_2)$	$g_1 \vee g_2$
$(g)\ \&\ A$	$R_{pre}(g) \rightarrow R_{pre}(A)$	$(g)\ \&\ R_{post}(A)$

provided *the condition g evaluates state components s_i,…,s_j, and OP_s_i is the schema name for the state component s_i.*

R_{mrg}

Definition 4.3.2 (R_{mrg}). *A $R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2))$ function is defined to merge the rewriting prefixes of A_1 and A_2 into one final prefix. Basically, it is equal to $R_{pre}(A_1) \rightarrow R_{pre}(A_2)$ if each state component retrieve schema expression in $R_{pre}(A_2)$, $\big(OP\_s_i\big)$, is* *different from any in $R_{pre}(A_1)$. However, for any state component retrieve schema expression in $R_{pre}(A_2)$, if it is the same as that in $R_{pre}(A_1)$, it is removed from $R_{pre}(A_2)$ before combination. For example,*

$$\begin{aligned}
&R_{mrg}\left(\left(OP\_x\right),\left(OP\_y\right)\right) = \left(OP\_x\right) \rightarrow \left(OP\_y\right)\\
&R_{mrg}\left(\left(OP\_x\right),\left(OP\_y\right) \rightarrow \left(OP\_x\right)\right) = \left(OP\_x\right) \rightarrow \left(OP\_y\right)
\end{aligned}$$

The R_{mrg} is easily extended to have multiple parameters but still keeps the similar definition.

$$\begin{aligned}
&R_{mrg}\left(\left(OP\_x\right),\left(OP\_y\right),\left(OP\_z\right)\right) = \left(OP\_x\right) \rightarrow \left(OP\_y\right) \rightarrow \left(OP\_z\right)\\
&R_{mrg}\left(\left(OP\_x\right),\left(OP\_y\right),\left(OP\_x\right)\right) = \left(OP\_x\right) \rightarrow \left(OP\_y\right)
\end{aligned}$$

Schema Expression as Action

R_{wrt} **Rule 25 (Schema Expression as Action).** In *Circus*, the schema which has the same name as a schema expression as action should be an operational schema. In other words, the schema shall include all state variables after operation v' (usually before operation v as well, but not strictly required because the initial schema may only include v' and exclude v) and possibly input and output variables (*ins*? and *outs*!), and it specifies a relationship between input and output variables, the state before operation, and the state after operation. The straight way to turn a schema into an operation is to include both the state schema (*State*) and its dashed version (*State'*) into its declarations by Δ and Ξ.

The rewrite of a schema expression as action is simply itself.

$$R_{wrt}\left(\begin{array}{l} SExp = [decl\ ;\ ins? : T_i\ ;\ outs! : T_o \mid pred] \\ \left(SExp\right) \end{array}\right)$$
$$= \left\{\begin{array}{l} SExp = [decl\ ;\ ins? : T_i\ ;\ outs! : T_o \mid pred] \\ \left(SExp\right) \end{array}\right.$$

□

Basic Actions

R_{wrt} **Rule 26 (Basic Actions).**

$$\begin{aligned}
R_{wrt}\,(\mathbf{Skip}) &= \mathbf{Skip}\\
R_{wrt}\,(\mathbf{Stop}) &= \mathbf{Stop}\\
R_{wrt}\,(\mathbf{Chaos}) &= \mathbf{Chaos}
\end{aligned}$$

□

Prefixing Action

R_{wrt} **Rule 27 (Prefixing Action).** For the communication without messages,

$$R_{wrt}\,(c \rightarrow A) = c \rightarrow R_{wrt}\,(A)$$

For the communication with output messages, if its expressions e does not evaluate state variables

$$R_{wrt}\,(c.e \rightarrow A) = c.e \rightarrow R_{wrt}\,(A)$$

If its expressions e evaluate state components $s_i, \cdots, s_j$, then

$$R_{wrt}\,(c.e \to A) = \left(Op\_s_i\right) \to \cdots \to \left(Op\_s_j\right) \to c.e \to R_{wrt}\,(A)$$

For the communication with input variables only,

$$R_{wrt}\,(c?x \to A(x)) = c?x \to R_{wrt}\,(A(x))$$
$$R_{wrt}\,(c?x : P \to A) = c?x : P \to R_{wrt}\,(A(x))$$

For the communication with multiple fields, provided e_1 and e_2 evaluate state variable $s_i, \cdots, s_j$, then

$$\begin{aligned}&R_{wrt}\,(c!e_1?x!e_2?y \to A(x,y))\\ &= \left(Op\_s_i\right) \to \cdots \to \left(Op\_s_j\right) \to c!e_1?x!e_2?y \to R_{wrt}\,(A(x,y))\end{aligned}$$

□

Guarded Action

R_{wrt} **Rule 28 (Guarded Action).** The definitions of R_{pre} and R_{post} are given in Definition 4.3.1, and that of R_{mrg} is given in Definition 4.3.2.

$$R_{wrt}\,\big((g) \mathbin{\&} A\big) = R_{mrg}\,(R_{pre}\,(g)\,, R_{pre}\,(A)) \to \big((g) \mathbin{\&} R_{post}\,(A)\big)$$

For example, if x and y are state variables,

$$\begin{aligned}&R_{wrt}\,\big((x > 0 \land y < 5) \mathbin{\&} c!x?z \to \mathbf{Skip}\big)\\ &= R_{mrg}\,\Big(\big(OP\_x\big) \to \big(OP\_y\big), \big(OP\_x\big)\Big) \to \big((x > 0 \land y < 5) \mathbin{\&} c!x?z \to \mathbf{Skip}\big)\\ &= \big(OP\_x\big) \to \big(OP\_y\big) \to \big((x > 0 \land y < 5) \mathbin{\&} c!x?z \to \mathbf{Skip}\big)\end{aligned}$$

□

Sequential Composition

R_{wrt} **Rule 29 (Sequential Composition).** A sequential composition of two actions is rewritten by rewriting both actions individually but taking the prefix of the rewrite of the first action $R_{pre}\,(A_1)$, out of the sequential composition.

$$R_{wrt}\,(A_1\,;\,A_2) = R_{pre}\,(A_1) \to (R_{post}\,(A_1)\,;\,R_{wrt}\,(A_2))$$

□

External Choice

R_{wrt} **Rule 30 (External Choice).**

$$R_{wrt}\,(AA_1 \mathbin{\Box} AA_2) = R_{mrg}\,(R_{pre}(AA_1), R_{pre}(AA_2)) \to (R_{post}\,(AA_1) \mathbin{\Box} R_{post}\,(AA_2))$$

where AA is a prefixed action defined in Definition B.3.1, or AA_1 and AA_2 are mutually exclusive guarded actions (see Φ Rule 28). □

Internal Choice

R_{wrt} **Rule 31 (Internal Choice).**

$$R_{wrt}\,(A_1 \sqcap A_2) = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \rightarrow (R_{post}\,(A_1) \sqcap R_{post}\,(A_2))$$

□

Parallel Composition and Interleaving

R_{wrt} **Rule 32 (Parallel Composition and Interleaving).** Parallel composition and interleaving have the similar rule.

$$\begin{aligned} & R_{wrt}\,(A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2) \\ & \quad = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \rightarrow (R_{post}\,(A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, R_{post}\,(A_2)) \\ & R_{wrt}\,(A_1 \,|\!|\!\lbrack\, ns_1 \mid ns_2 \,\rbrack\!|\!|\, A_2) \\ & \quad = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \rightarrow (R_{post}\,(A_1) \,|\!|\!\lbrack\, ns_1 \mid ns_2 \,\rbrack\!|\!|\, R_{post}\,(A_2)) \end{aligned}$$

□

Hiding

R_{wrt} **Rule 33 (Hiding).**

$$R_{wrt}\,\big(A \setminus cs\big) = R_{pre}\,(A) \rightarrow \big(R_{post}\,(A) \setminus cs\big)$$

□

Recursion

R_{wrt} **Rule 34 (Recursion).** The rewrite of a recursion $\mu X \bullet A\,(X)$ is the rewrite of its action part $A(X)$ and for each reference to X in A, it remains the same X (that is different from the action invocation rule R_{wrt} Rule 35). In addition, the recursion is partially restricted to appear as one of actions in an external choice, which is given in Definition B.3.1. The restriction states that if a recursion is one action in an external choice, the action of the recursion should be a prefixed action and its initial events shall not evaluate state variables.

$$R_{wrt}\,(\mu X \bullet A\,(X)) = \mu X \bullet R_{wrt}\,(A(X)) \qquad \text{and} \qquad R_{wrt}(X) = X$$

□

The reason for the restriction of recursion as one action in the external choice is because the additional state retrieve schema expressions added before the action may cause inappropriate resolution of the external choice. We have considered three possible solutions as presented below. Finally, this restriction makes it easy to be rewritten and still with the same behaviour.

In order to explain the potential problems of these solutions, the external choice below is a good example.

$$(\mu X \bullet (c!s_1 \rightarrow X)) \,\square\, (d!s_2 \rightarrow \mathbf{Skip})$$

Possible solution one If we define the rule for the recursion as

$$R_{wrt}(\mu X \bullet A(X)) = R_{pre}(A(x)) \rightarrow (\mu X \bullet R_{post}(A(X)))$$

then

$$\begin{aligned}
&R_{wrt}((\mu X \bullet (c!s_1 \rightarrow X)) \Box (d!s_2 \rightarrow \textbf{Skip}))\\
&= R_{mrg}(R_{pre}(c!s_1 \rightarrow X), R_{pre}(d!s_2 \rightarrow \textbf{Skip})) \rightarrow\\
&\qquad ((\mu X \bullet R_{post}(c!s_1 \rightarrow X)) \Box R_{post}(d!s_2 \rightarrow \textbf{Skip}))\\
&\hfill [\text{Rewriting rules for external choice and recursion}]\\
&= R_{mrg}\Big(\big(OP\_s_1\big), \big(OP\_s_2\big)\Big) \rightarrow ((\mu X \bullet (c!s_1 \rightarrow X)) \Box (d!s_2 \rightarrow \textbf{Skip}))\\
&\hfill [\text{Rewriting rules for prefixing}]\\
&= \big(OP\_s_1\big) \rightarrow \big(OP\_s_2\big) \rightarrow ((\mu X \bullet (c!s_1 \rightarrow X)) \Box (d!s_2 \rightarrow \textbf{Skip}))\\
&\hfill [R_{mrg} \text{ definition}]
\end{aligned}$$

Then the recursion might be expanded to $(c!s_1 \rightarrow (\mu X \bullet c!s_1 \rightarrow X))$. Therefore, the first s_1 has got the up-to-date value of s_1 by $\big(OP\_s_1\big)$, but the second s_1 within the recursion still uses the value from $\big(OP\_s_1\big)$ and cannot see the latest update to s_1 (which may be caused by other actions). That is a problem.

Possible solution two If we define the rule for the recursion as

$$R_{wrt}(\mu X \bullet A(X)) = \mu X \bullet R_{wrt}(A(x))$$

then

$$\begin{aligned}
&R_{wrt}((\mu X \bullet (c!s_1 \rightarrow X)) \Box (d!s_2 \rightarrow \textbf{Skip}))\\
&= R_{pre}(d!s_2 \rightarrow \textbf{Skip}) \rightarrow ((\mu X \bullet R_{wrt}(c!s_1 \rightarrow X)) \Box R_{post}(d!s_2 \rightarrow \textbf{Skip}))\\
&\hfill [\text{Rewriting rules for external choice and recursion}]\\
&= \big(OP\_s_2\big) \rightarrow \Big(\Big(\mu X \bullet \Big(\big(OP\_s_1\big) \rightarrow c!s_1 \rightarrow X\Big)\Big) \Box (d!s_2 \rightarrow \textbf{Skip})\Big)\\
&\hfill [\text{Rewriting rules for prefixing}]
\end{aligned}$$

Then the first construct in the action of the recursion is $\big(OP\_s_1\big)$ which is linked to an event in CSP and might resolve the external choice. However, it is not what we expect that the external choice is only resolved by the event c or d.

Possible solution three If we expand the recursion one more step and lead to $A(A(x))$, then use the solution one

$$R_{wrt}(\mu X \bullet A(X)) = R_{pre}(A(A(x))) \rightarrow (\mu X \bullet R_{post}(A(A(X))))$$

then

$$\begin{aligned}
&R_{wrt}((\mu X \bullet (c!s_1 \rightarrow X)) \Box (d!s_2 \rightarrow \textbf{Skip}))\\
&= R_{mrg}(R_{pre}(c!s_1 \rightarrow c!s_1 \rightarrow X), R_{pre}(d!s_2 \rightarrow \textbf{Skip})) \rightarrow\\
&\qquad ((\mu X \bullet R_{post}(c!s_1 \rightarrow c!s_1 \rightarrow X)) \Box R_{post}(d!s_2 \rightarrow \textbf{Skip}))\\
&\hfill [\text{Rewriting rules for external choice and recursion}]\\
&= R_{mrg}\Big(\big(OP\_s_1\big), \big(OP\_s_2\big)\Big) \rightarrow\\
&\qquad \Big(\Big(\mu X \bullet \Big(c!s_1 \rightarrow \big(OP\_s_1\big) \rightarrow c!s_1 \rightarrow X\Big)\Big) \Box (d!s_2 \rightarrow \textbf{Skip})\Big)\\
&\hfill [\text{Rewriting rules for prefixing}]
\end{aligned}$$

$$= \left(OP\_s_1\right) \rightarrow \left(OP\_s_2\right) \rightarrow$$
$$\left(\left(\mu X \bullet \left(c!s_1 \rightarrow \left(OP\_s_1\right) \rightarrow c!s_1 \rightarrow X\right)\right) \Box \left(d!s_2 \rightarrow \mathbf{Skip}\right)\right)$$

[R_{mrg} definition]

This has the similar problem as the possible solution one due to the fact that a step further unroll of the recursion will still result in consecutive $c!s_1$ events without $\left(OP\_s_1\right)$ between them.

Action Invocation

R_{wrt} **Rule 35 (Action Invocation).** The rewrite of an action invocation is the rewrite of the body of this action if the action name does not occur in its body.

$$R_{wrt}\left(A\right) = R_{wrt}\left(B(A)\right)$$

Where B is a function given in Definition B.2.15.

However, if the action name appears in its action body, then it is an implicit recursion. The application of the rewrite rule above will lead to infinite loop. Therefore, the rule is revised for this case.

$$R_{wrt}\left(A\right) = R_{wrt}\left(\mu A \bullet (B(A))\right)$$

According to R_{wrt} Rule 34, the action invocation A in $B(A)$ is rewritten to A and will not be expanded further. □

Unnamed Parametrised Action Invocation

R_{wrt} **Rule 36 (Unnamed Parametrised Action Invocation).** The rewrite of an unnamed parametrised action invocation is the rewrite of the body of the action with substituting e for each free occurrence of x in A.

$$R_{wrt}\left((x : T \bullet A)\,(e)\right) = R_{wrt}\left(A[e/x]\right)$$

□

Parametrised Action Invocation

R_{wrt} **Rule 37 (Parametrised Action Invocation).** Provided A is a parametrised action defined below,

$$PA \mathrel{\widehat=} x : T \bullet A$$

then the rewrite of the invocation of PA is the rewrite of its body with substituting e for each free occurrence of x in A.

$$R_{wrt}\left(PA(e)\right) = R_{wrt}\left((B(PA))\,(e)\right) = R_{wrt}\left(A[e/x]\right)$$

□

Iterated Operators

R_{wrt} **Rule 38 (Iterated Operators).** The rewrite of iterated operators is simple.

$$R_{wrt}\left(;\, x : T \bullet A(x)\right) = R_{pre}\left(A(x)\right) \rightarrow \left(;\, x : T \bullet R_{post}\left(A(x)\right)\right)$$
$$R_{wrt}\left(\Box\, x : T \bullet AA(x)\right) = R_{pre}\left(AA(x)\right) \rightarrow \left(\Box\, x : T \bullet R_{post}\left(AA(x)\right)\right)$$
$$R_{wrt}\left(\sqcap\, x : T \bullet A(x)\right) = R_{pre}\left(A(x)\right) \rightarrow \left(\sqcap\, x : T \bullet R_{post}\left(A(x)\right)\right)$$
$$R_{wrt}\left(\llbracket cs \rrbracket\; x : T \;\llbracket ns \rrbracket \bullet A(x)\right) = R_{pre}\left(A(x)\right) \rightarrow \left(\llbracket cs \rrbracket\; x : T \;\llbracket ns \rrbracket \bullet R_{post}\left(A(x)\right)\right)$$
$$R_{wrt}\left(|||\; x : T \;\|[ns]\| \bullet A(x)\right) = R_{pre}\left(A(x)\right) \rightarrow \left(|||\; x : T \;\|[ns]\| \bullet R_{post}\left(A(x)\right)\right)$$

□

Assignment

R_{wrt} **Rule 39 (Assignment).** A set of state variables s_l and local variables l_l can be updated simultaneously by an assignment. It is rewritten to an extra schema expression action

$$R_{wrt}\,(s_l, l_l := e_s, e_l) = R_{wrt}\left(\left(P\_assOp\right)\right)$$

and a corresponding schema in Z, that has before-state local variables as input and after-state local variables as output, to specify after-state state and local variables in its predicate.

$$\begin{aligned} P\_assOp == [\,& \Delta P\_StPar\ ;\ l_r? : T_{l_r?}\ ;\ l_l! : T_{l_l!} \mid \\ & P\_s_r' = (e_s[l_r?/l_r]) \land l_l! = (e_l[l_r?/l_r]) \land u' = u\,] \end{aligned}$$

where s_l and l_l in the left hand side of the assignment are renamed to dashed state variables s_l' and output variables $l_l!$ in the schema respectively, and local variables l_r that appear in the right hand side of the assignment in expressions e_s and e_l are renamed to $l_r?$; P is the name of the process; u denotes a set of all state variables that are not included in s_l.

By this rule, the state and local variables in the assignment are specified in Z. Therefore, $R_{pre}\,(x := e)$ is empty and $R_{post}\,(x := e) = R_{wrt}\,(x := e)$. □

Alternation

R_{wrt} **Rule 40 (Alternation).** For an alternation, provided it has n guarded actions, then it is rewritten to an external choice of 2^n guarded actions which can be further rewritten by the external choice R_{wrt} Rule 31 and the guarded action R_{wrt} Rule 28.

$$R_{wrt}\begin{pmatrix} \textbf{if} & g_1 \longrightarrow A_1 \\ [\!] & g_2 \longrightarrow A_2 \\ [\!] & \cdots \\ [\!] & g_n \longrightarrow A_n \\ \textbf{fi} & \end{pmatrix}$$

$$= R_{wrt}\begin{pmatrix} & (\neg g_1 \land \neg g_2 \land \cdots \land \neg g_n)\ \&\ \textbf{Chaos} \\ \Box & (g_1 \land \neg g_2 \land \cdots \land \neg g_n)\ \&\ A_1 \\ \Box & (g_1 \land g_2 \land \cdots \land \neg g_n)\ \&\ (A_1 \sqcap A_2) \\ \Box & \cdots \\ \Box & (\cdots \land g_i \cdots \land g_j \land \cdots \land g_k \land \cdots)\ \&\ (A_i \sqcap A_j \sqcap A_k) \\ \Box & \cdots \\ \Box & (g_1 \land g_2 \land \cdots \land g_n)\ \&\ (A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n) \end{pmatrix}$$

Among these guarded actions, their guarded conditions are all possible combination of original guarded conditions $g_1, \cdots, g_n$, and

- if all original guarded conditions are false, then it diverges,
- if only one original guarded condition is true and others are false, then it behaves like the guarded action, of which the guarded condition is true,
- if more than one guarded condition is true, then it behaves like an internal choice of all guarded actions of which the guarded conditions are true.

□

Table 4.1: Comparison of Alternation and Three Attempts

g_1	g_2	Alternation	Att1	Att2	Att3
true	true	$A_1 \sqcap A_2$	$A_1 \sqcap A_2$	$A_1 \Box A_2$	$A_1 \sqcap A_2$
true	false	A_1	$A_1 \sqcap \mathbf{Stop}$	A_1	A_1
false	true	A_2	$A_2 \sqcap \mathbf{Stop}$	A_2	A_2
false	false	**Chaos**	**Chaos**	**Chaos**	**Chaos**

For example,

$$
\begin{aligned}
& R_{wrt}\left(\begin{array}{ll} \mathbf{if} & x > y \longrightarrow c.y \to \mathbf{Skip} \\ [] & x < y \longrightarrow c.x \to \mathbf{Skip} \\ \mathbf{fi} & \end{array}\right) \\
& = R_{wrt}\left(\begin{array}{ll} & (\neg(x > y) \land \neg(x < y)) \mathbin{\&} \mathbf{Chaos} \\ \Box & ((x > y) \land \neg(x < y)) \mathbin{\&} c.y \to \mathbf{Skip} \\ \Box & (\neg(x > y) \land (x < y)) \mathbin{\&} c.x \to \mathbf{Skip} \\ \Box & ((x > y) \land (x < y)) \mathbin{\&} (c.y \to \mathbf{Skip} \sqcap c.x \to \mathbf{Skip}) \end{array}\right)
\end{aligned}
$$

[R_{wrt} Rule 40]

Considerations Three solutions have been attempted to rewrite the alternation. The informal description of each solution is shown as follows, where informal means not all notations used are valid *Circus* syntax. For instance, $\lhd\rhd$ is a UTP notation but it is used here merely to simplify the description, though it can be converted to the *Circus* notation.

The first solution is

$$\mathbf{Chaos} \lhd \neg g_1 \land \neg g_2 \rhd \left(\begin{array}{ll} & (g_1) \mathbin{\&} A_1 \\ \sqcap & (g_2) \mathbin{\&} A_2 \end{array}\right)$$

The second one is

$$\mathbf{Chaos} \lhd \neg g_1 \land \neg g_2 \rhd \left(\begin{array}{ll} & (g_1) \mathbin{\&} A_1 \\ \Box & (g_2) \mathbin{\&} A_2 \end{array}\right)$$

The third one is

$$\left(\begin{array}{ll} & (\neg g_1 \land \neg g_2) \mathbin{\&} \mathbf{Chaos} \\ \Box & (g_1 \land \neg g_2) \mathbin{\&} A_1 \\ \Box & (\neg g_1 \land g_2) \mathbin{\&} A_2 \\ \Box & (g_1 \land g_2) \mathbin{\&} A_1 \sqcap A_2 \end{array}\right)$$

A comparison of all these attempts with the original alternation is displayed in Table 4.1. From this table, we conclude that only the Att3 (the third solution) preserves the semantics after rewrite. This is also the reason we choose the third solution in our rewriting rule.

Variable Block

R_{wrt} **Rule 41 (Variable Block).**

$$R_{wrt}\,(\mathbf{var}\, x : T \bullet A) = R_{pre}\,(A) \to (\mathbf{var}\, x : T \bullet R_{post}\,(A))$$

$\Box$

Parametrisation by Value, by Result, and by Value-Result

R_{wrt} **Rule 42 (Parametrisation by Value, by Result, and by Value-Result).** The invocation of a parametrisation by value is rewritten to a variable block in which its declared variables x are initialised to the value of its actual parameters e in its action, while the invocation of parametrisation by result is rewritten to a variable block in which the actual parameters y are set to the value of its declared variables x in its action. Finally the invocation of a parametrisation by value-result, which has the combined effect of parametrisation by value and parametrisation by result, is rewritten to a variable block in which its declared variables x are initialised to the value of its actual parameters y and in the end y are set to the value of x in its action.

$$\begin{aligned}
&R_{wrt}\left((\mathbf{val}\, x : T \bullet A)\,(e)\right) = R_{wrt}\left(\mathbf{var}\, x : T \bullet (x := e \,;\, A)\right)\\
&R_{wrt}\left((\mathbf{res}\, x : T \bullet A)\,(y)\right) = R_{wrt}\left(\mathbf{var}\, x : T \bullet (A \,;\, y := x)\right)\\
&R_{wrt}\left((\mathbf{vres}\, x : T \bullet A)\,(y)\right) = R_{wrt}\left(\mathbf{var}\, x : T \bullet (x := y \,;\, A \,;\, y := x)\right)
\end{aligned}$$

□

Specification Statement, Assumption, and Coercion

R_{wrt} **Rule 43 (Specification Statement).** For a specification statement like $w : [\,pre, post\,]$, provided the frame w is composed of state variables s_w and local variables l_w, pre and $post$ contain free occurrences of before-state local variables l_b, and $post$ contains free occurrences of after-state local variables l_a', it is rewritten to a schema expression as action

$$R_{wrt}\left(w : [\,pre, post\,]\right) = R_{wrt}\left(\left(specOp\right)\right)$$

and an additional schema

$$\begin{aligned}
specOp ==[\,&\Delta P\_StPar \,;\, l_b? : T_{l_b} \,;\, l_a! : T_{l_a} \mid\\
&\left(pre[l_b?/l_b] \land \exists\, u' : T_u \bullet post[l_b?/l_b, l_a!/l_a']\right) \land s_u' = s_u\,]
\end{aligned}$$

in Z, where u denotes the variables in $post$ but not in the frame, and s_u denotes all state variables that are not in the s_w. □

For instance, if a process has three state components: s_1, s_2, and s_3, and a specification statement

$$\left(s_1, l_1 : [\,s_2 > 0 \land l_1 < 5, s_1' = s_2 + 3 \land l_1' = s_3 \land s_2' = 0 \land l_2' = l_1 + 1\,]\right)$$

where l_1 and l_2 are two local variables in scope, is rewritten to a schema expression action

$$\begin{aligned}
&R_{wrt}\left(s_1, l_1 : [\,s_2 > 0 \land l_1 < 5, s_1' = s_2 + 3 \land l_1' = s_3 \land s_2' = 0 \land l_2' = l_1 + 1\,]\right)\\
&= R_{wrt}\left(\left(specOp\right)\right) && [R_{wrt} \text{ Rule 43}]
\end{aligned}$$

and $specOp$ is a schema added to Z part

$$\begin{aligned}
specOp ==[\,&\Delta P\_StPar \,;\, l_1? : T_{l_1} \,;\, l_1! : T_{l_1} \mid\\
&(s_2 > 0 \land l_1? < 5) \land\\
&\left(\exists\, s_2' : T_{s_2} \,;\, l_2' : T_{l_2} \bullet s_1' = s_2 + 3 \land l_1! = s_3 \land s_2' = 0 \land l_2' = l_1? + 1\right) \land\\
&s_2' = s_2 \land s_3' = s_3\,]
\end{aligned}$$

R_{wrt} **Rule 44 (Assumption).** An assumption $\{pre\}$ is an abbreviation for the specification statement $:[\,pre, true\,]$. Therefore, it is rewritten to

$$R_{wrt}\,(\{pre\}) = R_{wrt}\left(\left(assmpOp\right)\right)$$

and an additional schema

$$assmpOp == [\,\Xi P\_StPar\,;\,l_b? : T_{l_b} \mid pre[l_b?/l_b]\,]$$

in Z. □

R_{wrt} **Rule 45 (Coercion).** A coercion $[\,post\,]$ is an abbreviation for the specification statement $:[\,true, post\,]$. Therefore, it is rewritten to

$$R_{wrt}\,([\,post\,]) = R_{wrt}\left(\left(coerOp\right)\right)$$

and an additional schema

$$coerOp == [\,\Xi P\_StPar\,;\,l_b? : T_{l_b} \mid \exists\, u' : T_u \bullet post[l_b?/l_b]\,]$$

in Z. □

Renaming

R_{wrt} **Rule 46 (Renaming).** An action renaming gives new names to the variables in the list and is rewritten to

$$R_{wrt}\,(A[v_{old} := v_{new}]) = R_{wrt}\,(A[v_{new}/v_{old}])$$

where all free occurrences of variables in v_{old} of A are substituted by corresponding variables in v_{new}. □

4.3.18 Generic Definitions

Generic definitions in *Circus* introduce a family of similar constructors. Our strategy for genericity in *Circus* is to rewrite generic constructors and finally without genericity afterwards. Generally, an instantiation of a generic definition is rewritten to a reference to a new definition which is obtained by substituting the actual parameters in the instantiation by the formal parameters. Finally the generic definition is removed.

The uniqueness of instantiations for different generic constructors is stated below.

- For generic axiomatic definitions, all instantiations, that have the same actual parameters and are from the same generic axiomatic definition, are regarded as constants that are from the same definition.
- For generic schemas, all instantiations, though that have the same schema name and actual parameters, are regarded as individually different schemas.
- For generic channels, all instantiations, that have the same channel name and actual parameters, are regarded as the same channel.
- For generic processes, all instantiations, though that have the same process name and actual parameters, are still regarded as individually different processes.

4.3.18.1 Generic Axiomatic Definitions

Generic axiomatic definitions are removed from the program but the instantiation of the axiomatic variables is rewritten.

R_{wrt} **Rule 47 (Generic Axiomatic Definitions).** For each instantiation of generic constants defined in a generic axiomatic definition, a corresponding axiomatic definition is added. This axiomatic definition is got from the generic axiomatic definition by substituting the actual parameters in the instantiation for the formal parameters in the generic definition. In addition, the generic constants are renamed in this new axiomatic definition as well. Consequently, the instantiations of all constants from this generic axiomatic definition with the same actual parameters become the reference to these new renamed constants. Finally, the original generic axiomatic definition is removed. □

For example, provided there is a generic axiomatic definition below, which defines two constants a and c.

$$
\begin{array}{l}
[X] \\
\hline
a : X \; ; c : \mathbf{P} \; X \\
\hline
a \in c
\end{array}
$$

And three instantiations: $a[T_1]$, $c[T_1]$, and $c[T_2]$, are in the *Circus* program. According to R_{wrt} Rule 47, for the first and second instantiation, because they have the same actual parameter T_1, one axiomatic definition shown below is added by replacing X in the generic axiomatic definition above by T_1, and both a and c are renamed as well. At the same time, the instantiations $a[T_1]$ and $c[T_1]$ are rewritten to a_{new_1} and c_{new_1}.

$$
\begin{array}{l}
a_{new_1} : T_1 \; ; c_{new_1} : \mathbf{P} \; T_1 \\
\hline
a_{new_1} \in c_{new_1}
\end{array}
$$

Then for the third instantiation $c[T_2]$, similarly an additional axiomatic definition is added and the instantiation is rewritten to c_{new_2}.

$$
\begin{array}{l}
a_{new_2} : T_2 \; ; c_{new_2} : \mathbf{P} \; T_2 \\
\hline
a_{new_2} \in c_{new_2}
\end{array}
$$

Additionally, the generic axiomatic definition is removed.

4.3.18.2 Generic Boxed and Horizontal Schemas

R_{wrt} **Rule 48 (Generic Schemas).** For each instantiation of generic schemas, a corresponding schema is added. This schema is got from the generic schema by substituting the actual parameters in the instantiation for the formal parameters in the generic schema. In addition, this schema is renamed and given a unique name. Consequently, the instantiation becomes the reference to this new renamed schema. Finally, the original generic schema is removed. □

For example, provided there is a generic schema defined below

$$
\begin{array}{l}
GenSch[X] \\
\hline
a : X \; ; c : \mathbf{P} \; X \\
\hline
a \in c
\end{array}
$$

and one of its instantiation $GenSch[T]$. According to R_{wrt} Rule 48, one schema Sch_{new} is added by replacing X in $GenSch$ by T, and the instantiation $GenSch[T]$ becomes Sch_{new}.

$$\begin{array}{|l}
\textit{Sch}_{new} \\ \hline
a : T\, ; c : \mathbb{P}\ T \\ \hline
a \in c \\ \hline
\end{array}$$

Furthermore, the generic schema is removed.

4.3.18.3 Generic Channel Declarations

A generic channel declaration introduces a family of channels.

R_{wrt} **Rule 49 (Generic Channel Declarations).** For the instantiations of channels from a generic channel declaration, if they have the same actual parameters, a corresponding channel declaration is added. This channel declaration is got from the generic channel declaration by substituting the actual parameters in the instantiations for the formal parameters in the generic channel declaration. In addition, the channels defined in this generic channel declaration are renamed and given a unique name. Consequently, the instantiations are rewritten to the references to these new renamed channels. Finally, the original generic channel declaration is removed. □

For example, there is a generic channel declaration below

$$\textbf{channel}[X] gin, gout : X \times X$$

and three instantiations: $gin[T_1]$, $gou[T_1]$ and $gou[T_2]$. For the first and second instantiation with the same actual parameter T_1, a corresponding channel declaration below is added. Additionally, the instantiations become the references to the new channels: gin_{new_1} and $gout_{new_1}$.

$$\textbf{channel}\ gin_{new_1}, gout_{new_1} : T_1 \times T_1$$

And for the third instantiation, since it has the different actual parameter, another corresponding channel declaration below is added. Additionally, the instantiation becomes the reference to the new channel: $gout_{new_2}$.

$$\textbf{channel}\ gin_{new_2}, gout_{new_2} : T_2 \times T_2$$

Eventually, the generic channel declaration is removed.

4.3.18.4 Generic Processes

R_{wrt} **Rule 50 (Generic Processes).** For each instantiation of generic processes, a corresponding process is added. This process is got from the generic process by substituting the actual parameters in the instantiation for the formal parameters in the generic process. In addition, this process is renamed and given a unique name. Consequently, the instantiation becomes the reference to this new renamed process. Finally, the original generic process is removed. □

For instance, suppose a generic process is declared below

$$\begin{array}{l}
\textbf{process}\ [X]\, GProc \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ State == [\, s : \mathbb{P}\ X \mid \#\ s < 5\,] \\
\quad Init == [\, State' \mid s' = \varnothing\,] \\
\quad \bullet\ \textbf{Skip} \\
\textbf{end}
\end{array}$$

and its instantiation $GProc[\mathbb{N}]$. According to R_{wrt} Rule 50, one process below is added and its instantiation becomes the reference to this new process: $GProc_{new}$.

$$
\begin{array}{l}
\mathbf{process}\ \ GProc_{new} \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ \ State == [\, s : \mathbf{P}\ \mathbb{N} \mid \#\ s < 5 \,] \\
\quad Init == [\, State' \mid s' = \varnothing \,] \\
\quad \bullet\ \mathbf{Skip} \\
\mathbf{end}
\end{array}
$$

Note: The instantiation of $GProc[\mathbb{N}]$ in different processes may refer to different processes and not the same one. But for the instantiation of generic channel declaration, it refers to the same channel.

4.4 *Circus* State Part to B - Ω

4.4.1 *Circus* State Part to ISO Standard Z - Ω_1

The function Ω_1 translates the state part in a rewritten *Circus* program to a Z specification in ISO Standard Z. Because the state part of *Circus* is also written in ISO Standard Z, for most constructs they are just a direct map without changes. However, a rewritten *Circus* program still has the same structure as the original program—all state components and schemas are encapsulated within the processes—but the state and schemas in a ISO Standard Z specification are flat. Therefore, we need to merge all state components and schemas into one global and flat specification in ISO Standard Z.

Definition 4.4.1 (Initialisation Schema). *In an explicitly defined process of Circus, there is a special notation* **state** *to mark the state schema of this process. However, there is no notation to identify the initialisation schema—it is no necessary too because the behaviour of this process is given by its main action. When all these basic processes are merged together to form an individually complete Z specification, the fact of no initialisation schemas will make all state variables in this new Z specification under no specific constraints except their types, which inevitably makes animation and model checking more difficult. To ease this problem, it is necessary to identify all initialisation schemas and finally initial state will be more specific. A schema Sch in a basic process is regarded as an initialisation schema if it complies with the rules defined below.*

- *The declaration part of the schema includes only all dashed state variables (v'), no before-state variables (v), and no other variables.*
- *The first action in the process's main action is a schema expression as action $\big(Sch\big)$.*
- *If there is one schema which meets two rules above, it is regarded as the initialisation schema. Otherwise, the initialisation schema will be like*

  $$Init == [\, (StPar)' \mid true \,]$$

  where the predicate on the dashed state variables is true.

Ω_1 **Rule 1 (States and Schemas Merge).** If there are more than one explicitly defined process, their states and operations are merged in the resultant Z specification. Assume there are n explicitly defined processes, named $P_1, P_2, \ldots, P_n$, in a *Circus* specification. Their states and schemas are merged as shown in Figure 4.4. The *state* schema is a conjunction of state schemas from all processes, as well as the *Init* schema, which is a conjunction of all initialisation schemas from all processes. All other schemas from each process will be translated to corresponding schemas with their own *declaration* and *predicate*. Additionally they shall keep state components from other processes unchanged by including Ξ of all other state paragraphs into their declaration.

$$
\begin{array}{l}
\Omega_1(\text{Rewritten } \textit{Circus} \text{ Program}) \\
= \Omega_1 \left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{process}\ P_1 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, P_1\_StPar == [\, P_1\_s_1 : T_{11} ; \cdots \\
\qquad P_1\_s_{m_1} : T_{1m_1} \mid ps_1 \,] \\
\quad P_1\_Init == [\, (P_1\_StPar)' \mid pi_1 \,] \\
\quad P_1\_Pars == [\, decl_1 \mid p_1 \,] \\
\quad \bullet\ A \\
\textbf{end}
\end{array} \right) \\
\cdots \\
\left(\begin{array}{l}
\textbf{process}\ P_n \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, P_n\_StPar == [\, P_n\_s_1 : T_{n1} ; \cdots \\
\qquad P_n\_s_{m_n} : T_{nm_n} \mid ps_n \,] \\
\quad P_n\_Init == [\, (P_n\_StPar)' \mid pi_n \,] \\
\quad P_n\_Pars == [\, decl_n \mid p_n \,] \\
\quad \bullet\ A \\
\textbf{end}
\end{array} \right)
\end{array} \right) \\
= \left(\begin{array}{l}
P_1\_StPar == [\, P_1\_s_1 : T_{11} ; \cdots P_1\_s_{m_1} : T_{1m_1} \mid ps_1 \,] \\
\ldots \\
P_n\_StPar == [\, P_n\_s_1 : T_{n1} ; \cdots P_n\_s_{m_n} : T_{nm_n} \mid ps_n \,] \\
State == P_1\_StPar \land \cdots \land P_n\_StPar \\
Init == [\, (State)' \mid pi_1 \land \cdots \land pi_n \,] \\
P1\_Pars == \left[\begin{array}{l} P_1\_Pars.decl_1 ; \Xi P_2\_StPar ; \ldots ; \Xi P_n\_StPar \\ \qquad \mid P_1\_Pars.p_1 \end{array} \right] \\
\ldots \\
Pn\_Pars == \left[\begin{array}{l} P_n\_Pars.decl_n ; \Xi P_1\_StPar ; \ldots ; \Xi P_{n-1}\_StPar \\ \qquad \mid P_n\_Pars.p_n \end{array} \right]
\end{array} \right)
\end{array}
$$

Figure 4.4: Ω_1 Function

4.4.2 ISO Standard Z to ZRM - Ω_2

The function Ω_2 takes the constructs in ISO Standard Z as input and outputs the corresponding constructs in ZRM. It is only syntactical transformation. The differences between them are investigated in CADiZ [87].

4.4.2.1 Specification Structure

The specification in ISO Standard Z is grouped by sections, however it is not the case in ZRM. A specification in ZRM is a LaTeX document as a whole. Thus it shall include basic structure of a LaTeX document.

Ω_2 **Rule 1 (Structure).** When translating a ISO Standard Z specification to ZRM, it adds the following lines in the beginning of document,

```
\documentclass{article}
\usepackage{fuzz}

\begin{document}
```

and appends the line below in the end of document.

```
\end{document}
```

□

4.4.2.2 Sections

In ZRM, if one specification tends to use existing libraries or specifications, it has to copy all of them into its specification, which makes it every difficult to reuse them and maintain the updates. ISO Standard Z introduces a *section* notation to address this toolkit reuse problem in ZRM. One section can reuse other sections by references to them very conveniently in its *section header*. Below is an example of a section header that declares a section named *csection* reusing *circus_toolkit*.

section *csection* **parents** *circus_toolkit*

We only consider one section for each rewritten *Circus* specification because the original multiple sections have been rewritten by R_{wrt} Rule 2 and our rule is very simple by removing the section header declaration.

Ω_2 **Rule 2 (Sections).** A section declaration in ISO Z Standard

```
\begin{zsection}
    \SECTION\ secname \parents\ parentsecname
\end{zsection}
```

is simply removed or commented when translated to ZRM.

```
%\begin{zsection}
%    \SECTION\ secname \parents\ parentsecname
%\end{zsection}
```

4.4.2.3 Mutually Recursive Free Types

Mutually recursive free type is introduced in ISO Standard Z but not supported in ZRM. We restrict its usage in *Circus* to facilitate the translation though it can semantically transform to type definition, membership, total functionality, injectivity, disjointness and induction constraints [70, D.8.1].

Ω_2 **Rule 3 (Mutually recursive free type).** It is restricted to use in *Circus*. □

4.4.2.4 Horizontal Schema Definition

Ω_2 **Rule 4 (Horizontal Definition).** In ISO Standard Z, horizontal schema definition uses ==, while in ZRM it is $\widehat{=}$. Therefore, our rule is

$$\begin{aligned}&\Omega_2\,(Sch == [\,decl \mid pred\,])\\&= Sch \mathrel{\widehat{=}} [\,decl \mid pred\,]\end{aligned}$$

□

4.4.2.5 Schema Decoration

Ω_2 **Rule 5 (Schema Decoration).** Schema decoration is different: `S~'` or `(S)'` in ISO Standard Z and S' in ZRM.

Ω_2(`S~'`) = `S'`
Ω_2(`S '`) = `S'`
Ω_2(`(S)'`) = `S'`

□

4.4.2.6 Arithmetic Negation and Subtraction

Ω_2 Rule 6 (Arithmetic Negation). In ISO Standard Z, arithmetic negation and subtraction uses the different notations: `\negate` and `-` respectively. However, in ZRM, the same notation `-` is used for both. Thus we translate `\negate` to `-`.

$\Omega_2(\texttt{\textbackslash negate}) = \texttt{-}$ □

4.4.2.7 Singleton Set of Schema

Ω_2 Rule 7 (Singleton Set of Schema). $\{sch\}$, provided `sch` is a schema, is parsed differently as singleton set and set comprehension in ISO Standard Z and ZRM respectively. ZRM uses parenthesized expression $\{(sch)\}$.

$$\Omega_2(\{sch\}) = \{(sch)\}$$

provided *sch* is a schema. □

4.4.2.8 Lambda-expression, Mu-expression, and Local Definition

Ω_2 Rule 8 (Lambda-expression and Mu-expression). For lambda-expression, mu-expression, and local definition, ZRM requires they are parenthesized but parentheses in ISO Standard Z can be omitted. When they are translated, we add parentheses for them.

$\Omega_2(\mu\, x : T \mid P \bullet E) = (\mu\, x : T \mid P \bullet E)$
$\Omega_2(\lambda\, x : T \mid P \bullet E) = (\lambda\, x : T \mid P \bullet E)$
$\Omega_2(\mathbf{let}\, x == y \bullet E) = (\mathbf{let}\, x == y \bullet E)$

□

4.4.2.9 Segment Relation or Infix of sequence

Ω_2 Rule 9 (Segment Relation or Infix). In ISO standard Z, infix is used instead of in in ZRM.

$$\Omega_2(\text{infix}) = \mathsf{in}$$

□

4.4.2.10 Set Symmetric Difference

Ω_2 Rule 10 (Set Symmetric Difference). In ISO standard Z, $\ominus$ is an operator for symmetric difference of set however no counterpart exists in ZRM. We define it based on current operators: set union and set difference.

$$\Omega_2(s1 \ominus s2) = (s1 \setminus s2) \cup (s2 \setminus s1)$$

□

4.4.2.11 Boolean Type

Circus defines the boolean type $\mathbb{B}$ and two values: **True** and **False** in its prelude of `circus_toolkit`—actually they are defined as free type. However, they are not a valid syntax of both ISO Standard Z and ZRM dialects.

Ω_2 Rule 11 (Boolean Type). One free type below is added once in the header of the program when translated to ZRM.

$$\mathbb{B} ::= \mathbf{True} \mid \mathbf{False}$$

□

4.4.3 ZRM to B Machine - Ω_3

The target language of the link in our approach of model checking *Circus* is $CSP \parallel B$ and the soundness of the link, which is proved in Chapter 5, is also based on $CSP \parallel B$. Though $CSP \parallel_B Z$ is used previously for the combination of CSP and ZRM, its semantics is still defined on $CSP \parallel B$. Therefore, Z in ZRM is merely an intermediate representation of the final B specification. To translate Z in ZRM to B, the Ω_3 function is defined to link constructs in ZRM to their counterparts in B. The general principles of this translation are listed below.

- The state components, their types, and their predates in the state schema in Z is translated to variables in the `VARIABLES` clause, basic types in the `INVARIANT` clause, and constraints in the corresponding operations (that refer to this state schema) respectively.
- The *Init* schema in Z is translated to the `INITIALISATION` clause in which the constraints are a conjunction of the corresponding predicate in the *Init* schema and the predicate from the state schema.
- Each operational schema in Z is translated to an operation in which the constraints are a conjunction of the corresponding predicate in the operational schema and the predicate from the state schema.

In order to give the soundness of this translation from abstract data type in Z to that in B, we can reason it from three perspectives as follows.

- The state space in Z and that in B should be the same: the same number of state variables, the same types, and the same constraints applied.
- The initial state specified by the *Init* schema in Z and that by the `INITIALISATION` and `INVARIANT` clauses in B should be the same: the same constraints on the state space.
- Each operational schema in Z has a corresponding operation in B. They should have the same semantics in terms of the design theory of UTP: the same precondition and postcondition.

If all three rules are complied in the translation, we can conclude it is sound. To some extent, these rules also imply implementation of the translator.

Finally, an implementation of the Ω_3 function, which is adopted in this thesis, is the ProZ [88] in ProB. For brevity, the details of ProZ are not given in this thesis.

4.5 *Circus* Behaviour to CSP$_M$ and Z - Φ

The function Φ takes the behavioural part of a *Circus* specification as input and output the corresponding structures in CSP$_M$.

Most of rules given in this section are easily understood while only some of them which are shown below are not obvious.

- Schema Expression as Action: Φ Rule 21
- Simplified Schema Expression as Action: Φ Rule 22
- External Choice: Φ Rule 27
- External Choice (Mutual Exclusively Guarded Actions): Φ Rule 28
- Parallel Composition and Interleaving (Disjoint Variables in Scope): Φ Rule 30

- Parallel Composition and Interleaving (Disjoint Variables in Updating): Φ Rule 31
- Iterated External Choice: Φ Rule 36
- Variable Block: Φ Rule 38

4.5.1 Types, Expressions and Operators

Φ Rule 1 (Types, Expressions and Operators). The map of types, expressions and operators in *Circus* to CSP is displayed in Appendix D.2.

4.5.2 Abbreviation Definitions

Φ Rule 2 (Abbreviation Definitions).

$$\Phi(AbbrDef == exp) = AbbrDef = \Phi(exp)$$

□

4.5.3 Axiomatic Definitions

Φ Rule 3 (Axiomatic Definitions). According to R_{wrt} Rule 5, axiomatic definitions have been rewritten to only have one instance. These rewritten axiomatic definitions are linked to CSP by removing original predicate in *Circus* and keeping the instances. For example,

$$\begin{array}{|l}
size, max\_size : \mathbb{N} \\
\hline
size < max\_size \\
size = 5 \\
max\_size = 15
\end{array}$$

is rewritten to

$$\begin{array}{l}
size = 5 \\
max\_size = 15
\end{array}$$

□

4.5.4 Channel Declarations

Φ Rule 4 (Channel Declarations).

$$\begin{array}{lcl}
\Phi\,(\mathbf{channel}\ \ c_1, \cdots, c_n) & = & \mathrm{channel}\ c_1, \cdots, c_n \\
\Phi\,(\mathbf{channel}\ \ c_1, \cdots, c_n : T) & = & \mathrm{channel}\ c_1, \cdots, c_n : \Phi(T)
\end{array}$$

□

4.5.5 Channel Set Declarations

Φ Rule 5 (Channel Set Declarations). A channel set paragraph in *Circus* links a channel set name and a channel set expression. This is similar to the construct in CSP.

$$\Phi(\mathbf{channelset}\ N == CSExp) = \mathrm{N} = \Phi(CSExp)$$

□

4.5.6 Channel Set Expressions

Φ Rule 6 (Channel Set Expressions). There are three types of channel set expressions: empty channel set ($\{\!|\ |\!\}$), channel enumeration (enclosed between $\{\!|$ and $|\!\}$), and other expressions by set union, set intersection, set difference, set extension, reference and application expressions.

Basic channel set expressions, including empty channel set and channel enumeration, declare a set of all events associated with channels inside. For example, $\{\!|\ c, d\ |\!\}$ means all events associated with channels c and d, such as $c.1$, $c.2$, $c.3$ and d if c is declared as **channel** $c : 1..3$ and d as **channel** d. That is the exactly same as that in CSP. So the translation rule is straightforward.

$$\begin{array}{lcl}
\Phi(\{\!|\ |\!\}) & = & \{|\,|\} \\
\Phi(\{\!|\ c_1, c_2, \cdots, c_n\ |\!\}) & = & \{|c_1, c_2, \cdots, c_n|\}
\end{array}$$

For other channel set expressions, they are translated as *Circus* expressions to CSP by Φ Rule 1. Additionally, a reference to a channel set expression is the reference itself.

$$\begin{array}{lcl}
\Phi(\mathit{CSExp}_1 \cup \mathit{CSExp}_2) & = & \mathrm{union}(\Phi(\mathit{CSExp}_1), \Phi(\mathit{CSExp}_2)) \\
\Phi(\mathit{CSExp}_1 \cap \mathit{CSExp}_2) & = & \mathrm{inter}(\Phi(\mathit{CSExp}_1), \Phi(\mathit{CSExp}_2)) \\
\Phi(\mathit{CSExp}_1 \setminus \mathit{CSExp}_2) & = & \mathrm{diff}(\Phi(\mathit{CSExp}_1), \Phi(\mathit{CSExp}_2)) \\
\Phi(\mathit{CSRef}) & = & \mathrm{CSRef}
\end{array}$$

$\square$

4.5.7 Name Set

Φ Rule 7 (Name Set). A name set paragraph in *Circus* declares the name of the name set and its expression. The name set is used in parallel composition and interleaving of actions, but there is no corresponding construct in CSP. Therefore no translation rule is given. According to Φ Rule 30 and Φ Rule 32, name sets are used to rewrite parallel composition and interleaving. $\square$

4.5.8 Explicitly Defined Processes

Φ Rule 8 (Explicitly Defined Processes). For an explicitly defined process P, its main action is linked to a CSP process with the same name as this process. Its state schema and other Z paragraphs are linked to Z and finally to B by the Ω function.

$$\Phi\left(\begin{array}{l}
\mathbf{process}\ \ P \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, \mathit{StPar} == [\, \mathit{decl} \mid \mathit{pred}\,] \\
\quad \mathit{Pars} == [\cdots] \\
\quad \bullet\ A \\
\mathbf{end}
\end{array}\right) \;=\; P = \Phi(A)$$

where StPar, Pars, and A denote the state paragraph, Z paragraphs and the main action respectively. $\square$

4.5.9 Compound Processes

4.5.9.1 Sequential Composition

Φ Rule 9 (Sequential Composition).

$$\Phi\,(P_1\,;P_2) \;=\; \Phi\,(P_1)\,;\Phi\,(P_2)$$

$\square$

4.5.9.2 External Choice

Φ Rule 10 (External Choice).

$$\Phi\left(P_1 \Box P_2\right) \quad = \quad \Phi\left(P_1\right) \Box \Phi\left(P_2\right)$$

□

4.5.9.3 Internal Choice

Φ Rule 11 (Internal Choice).

$$\Phi\left(P_1 \sqcap P_2\right) \quad = \quad \Phi\left(P_1\right) \sqcap \Phi\left(P_2\right)$$

□

4.5.9.4 Parallel Composition

Φ Rule 12 (Parallel Composition).

$$\Phi\left(P_1 \llbracket\, cs \,\rrbracket P_2\right) \quad = \quad \Phi\left(P_1\right) \underset{\Phi(cs)}{\parallel} \Phi\left(P_2\right)$$

□

4.5.9.5 Interleaving

Φ Rule 13 (Interleaving).

$$\Phi\left(P_1 \mid\mid\mid P_2\right) \quad = \quad \Phi\left(P_1\right) \mid\mid\mid \Phi\left(P_2\right)$$

□

4.5.9.6 Hiding

Φ Rule 14 (Hiding).

$$\Phi\left(P \setminus cs\right) \quad = \quad \Phi\left(P\right) \setminus \Phi(cs)$$

□

4.5.9.7 Process Invocation

Φ Rule 15 (Process Invocation).

$$\Phi(P) \quad = \quad P$$

A reference to a process is given by the copy rule and it is the body of the process. □

4.5.9.8 Iterated Sequential Composition

Φ Rule 16 (Iterated Sequential Composition).

$$\Phi(;\, x : T \bullet P(x)) \quad = \quad ;_{x:\Phi(T)} \bullet \Phi(P(x))$$

provided T is a finite sequence. Here $P(x)$ actually is the rewriting of the parametrised process by R_{wrt} Rule 22. □

4.5.9.9 Iterated External Choice

Φ Rule 17 (Iterated External Choice).

$$\Phi(\Box\, x : T \bullet P(x)) \quad = \quad \Box_{x:\Phi(T)} \bullet \Phi(P(x))$$

provided T is a finite set. Here $P(x)$ actually is the rewriting of the parametrised process by R_{wrt} Rule 22. □

4.5.9.10 Iterated Internal Choice

Φ Rule 18 (Iterated Internal Choice).

$$\Phi(\sqcap\, x : T \bullet P(x)) \quad = \quad \sqcap_{x:\Phi(T)} \bullet \Phi(P(x))$$

provided T is a finite set and must not be empty. Here $P(x)$ actually is the rewriting of the parametrised process by R_{wrt} Rule 22. □

4.5.9.11 Iterated Parallel Composition

Φ Rule 19 (Iterated Paralllel Composition).

$$\Phi(\llbracket CS \rrbracket\, x : T \bullet P(x)) \quad = \quad \underset{CS\ x:\Phi(T)}{\|} \bullet \Phi(P(x))$$

provided T is a finite set. Here $P(x)$ actually is the rewriting of the parametrised process by R_{wrt} Rule 22. □

4.5.9.12 Iterated Interleaving

Φ Rule 20 (Iterated Interleaving).

$$\Phi(|||\, x : T \bullet P(x)) = |||_{x:\Phi(T)} \bullet \Phi(P(x))$$

provided T is a finite set. Here $P(x)$ actually is the rewriting of the parametrised process by R_{wrt} Rule 22. □

4.5.10 Actions

4.5.10.1 Schema Expression as Action

Schema expression as action is declared as an event and finally hidden. An additional *SExp_fOp* schema is enabled only if the precondition of *SExp* does not hold.

Φ Rule 21 (Schema Expression as Action). A schema expression as action $(SExp)$ is linked to an external choice of the same name event *SExp* with input and output variables, and another event *SExp_fOp* whose precondition is the negation of the precondition of *SExp*. Therefore, if the precondition of *SExp* holds, it engages *SExp* event; otherwise, it engages *SExp_fOp* event and consequently diverges as **div**. Finally, these events are hidden from communication by adding both events to HIDE_CSPB. That makes it semantically equal to the schema expression as action in *Circus*.

$$\Phi\left(\left(SExp\right)\right)$$
$$= \left\{\begin{array}{l} channel\ SExp : \Phi\left(T_i\right).\Phi\left(T_o\right) \\ channel\ SExp\_fOp : \Phi\left(T_i\right) \\ HIDE\_CSPB = \{\mid SExp, SExp\_fOp \mid\} \\ (SExp!ins?outs \rightarrow SKIP \Box SExp\_fOp!ins \rightarrow \mathbf{div}) \\ SExp\_fOp = [\Xi StPar\, ; ins? : T_i \mid \neg\, \mathbf{pre}\, SExp] \end{array}\right.$$

provided *SExp* is a schema in Z with input variables *ins*? and output variables *outs*!; *SExp_fOp* is an additional schema in Z; particularly, its predicate is the negation of the precondition of *SExp*.

If *ins* and *outs* in *SExp* schema are empty, the rule is simplified to

$$\Phi\left(\left(SExp\right)\right) = \begin{cases} \text{channel } SExp, SExp\_fOp \\ \text{HIDE\_CSPB}=\{\!|SExp, SExp\_fOp|\!\} \\ (SExp \rightarrow SKIP \mathrel{\Box} SExp\_fOp \rightarrow \mathbf{div}) \\ SExp\_fOp \mathrel{\widehat=} [\Xi StPar \mid \neg\, \mathbf{pre}\, SExp] \end{cases}$$

□

Φ Rule 22 (Simplified Schema Expression as Action). If the precondition of *SExp* always holds such as state component retrieve schema expressions and assignments, Φ Rule 21 is simplified because it is not possible to make its precondition be evaluated to *false*.

$$\Phi\left(\left(SExp\right)\right) = \begin{cases} \text{channel } SExp : \Phi\left(T_i\right).\Phi\left(T_o\right) \\ \text{HIDE\_CSPB}=\{\!|SExp|\!\} \\ \begin{cases} (SExp!ins?outs \rightarrow SKIP) & \text{if } \left(SExp\right) \text{ as process} \\ (SExp!ins?outs) & \text{if } \left(SExp\right) \text{ as communication} \end{cases} \end{cases}$$

If *ins* and *outs* in the *SExp* schema are empty, the rule is simplified to

$$\Phi\left(\left(SExp\right)\right) = \begin{cases} \text{channel } SExp \\ \text{HIDE\_CSPB}=\{\!|SExp|\!\} \\ \begin{cases} (SExp \rightarrow SKIP) & \text{if } \left(SExp\right) \text{ as process} \\ (SExp) & \text{if } \left(SExp\right) \text{ as communication} \end{cases} \end{cases}$$

□

4.5.10.2 CSP Actions

Basic Actions

Φ Rule 23 (Basic Actions).

$$\begin{aligned} \Phi\left(\mathbf{Stop}\right) &= STOP \\ \Phi\left(\mathbf{Skip}\right) &= SKIP \\ \Phi\left(\mathbf{Chaos}\right) &= \mathbf{div} \end{aligned}$$

□

Prefixing

Φ Rule 24 (Prefixing).

$$\begin{aligned} \Phi\left(c \rightarrow A\right) &= c \rightarrow \Phi\left(A\right) \\ \Phi\left(c.e \rightarrow A\right) &= c.\Phi\left(e\right) \rightarrow \Phi\left(A\right) \\ \Phi\left(c!e \rightarrow A\right) &= c!\Phi\left(e\right) \rightarrow \Phi\left(A\right) \\ \Phi\left(c?x \rightarrow A\left(x\right)\right) &= c?x \rightarrow \Phi\left(A(x)\right) \\ \Phi\left(c?x : pred \rightarrow A(x)\right) &= c?x : \{y \mid y \text{ <- } \Phi\left(T_c\right), \Phi\left(pred\right)\} \rightarrow \Phi\left(A(x)\right) \\ \Phi\left(\left(SExp\right) \rightarrow A\right) &= \Phi\left(\left(SExp\right)\right) \rightarrow \Phi\left(A\right) \end{aligned}$$

provided T_c is the type of channel c, and *pred* is the input restriction predicate of x. □

Guarded Action

Φ Rule 25 (Guarded Action).

$$\Phi\left((g) \mathbin{\&} A\right) \quad = \quad \Phi(g) \;\&\; \Phi(A)$$

$\square$

Sequential Composition

Φ Rule 26 (Sequential Composition).

$$\Phi(A_1 \mathbin{;} A_2) \quad = \quad \Phi(A_1) \mathbin{;} \Phi(A_2)$$

$\square$

External Choice

Φ Rule 27 (External Choice). External choice of actions in *Circus* is only resolved by external events of the process or termination. Internal state changes of the process, such as schema expression as action and assignment, would not resolve it. Thus we restrict the actions that can occur in external choice to AA.

$$\Phi(AA_1 \mathbin{\square} AA_2) \quad = \quad \Phi(AA_1) \mathbin{\square} \Phi(AA_2)$$

$\square$

Φ Rule 28 (External Choice (Mutual Exclusively Guarded Actions)). Provided both actions are guarded action and their conditions (g_1 and g_2) are mutually exclusive, that is, $g_1 = \neg g_2$, then their guarded actions are not restricted to prefixed actions.

$$\Phi\left((g_1) \mathbin{\&} A_1 \mathbin{\square} (g_2) \mathbin{\&} A_2\right) \quad = \quad \Phi\left((g_1) \mathbin{\&} A_1\right) \mathbin{\square} \Phi\left((g_2) \mathbin{\&} A_2\right)$$

$\square$

Internal Choice

Φ Rule 29 (Internal Choice).

$$\Phi(A_1 \sqcap A_2) \quad = \quad \Phi(A_1) \sqcap \Phi(A_2)$$

$\square$

Parallel Composition and Interleaving

Φ Rule 30 (Parallel Composition and Interleaving (Disjoint Variables in Scope)).

$$\begin{aligned}
\Phi(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) &= \Phi(A_1) \underset{\Phi(cs)}{\|} \Phi(A_2) \\
\Phi(A_1 \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} A_2) &= \Phi(A_1) \mathbin{|||} \Phi(A_2)
\end{aligned}$$

provided

$$\begin{aligned}
ns_1 &= scpV(A_1) \\
ns_2 &= scpV(A_2)
\end{aligned}$$

where $scpV$ (Definition B.2.9) is a function to get a set of all variables in scope in an action. $\square$

Φ Rule 31 (Parallel Composition and Interleaving (Disjoint Variables in Updating)).

$$\begin{aligned}\Phi\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right) &= \Phi\left(A_1\right) \underset{\Phi(cs)}{\parallel} \Phi\left(A_2\right)\\ \Phi\left(A_1 \,|\!|\!|[ns_1 \mid ns_2]|\!|\!|\, A_2\right) &= \Phi\left(A_1\right) \,|\!|\!|\, \Phi\left(A_2\right)\end{aligned}$$

provided

$$\begin{aligned}&wrtV\left(A_1\right) = ns_1\\ &wrtV\left(A_2\right) = ns_2\\ &wrtV\left(A_1\right) \cap scpV\left(A_2\right) = \varnothing\\ &wrtV\left(A_2\right) \cap scpV\left(A_1\right) = \varnothing\end{aligned}$$

where *wrtV* (Definition B.2.5) is a function to get a set of variables written in an action.

The assumption states that 1) all variables to be written in A_1 and A_2 are in its own partition and consequently it is not necessary to discard any variables after termination, and 2) all variables to be written in one action are not seen by another action which makes it safe to update these variables. □

Φ Rule 32 (Parallel Composition and Interleaving). Except Φ Rule 30 and Φ Rule 31, the general parallel composition and interleaving of actions are not supported. □

Hiding

Φ Rule 33 (Hiding).

$$\Phi\left(A \setminus cs\right) \;=\; \Phi(A) \setminus \Phi(cs)$$

□

Recursion

Φ Rule 34 (Recursion).

$$\Phi\left(\mu X \bullet A(X)\right) \;=\; \text{let X} = \Phi\left(A(X)\right) \text{ within X} \quad \text{and} \quad \Phi\left(X\right) = X$$

□

Iterated Sequential Composition

Φ Rule 35 (Iterated Sequential Composition).

$$\Phi\left(;\, x : T \bullet A(x)\right) \;=\; ;_{x:\Phi(T)} \bullet \Phi\left(A(x)\right)$$

provided T is a finite sequence. □

Iterated External Choice

Φ Rule 36 (Iterated External Choice).

$$\Phi\left(\Box\, x : T \bullet AA(x)\right) \;=\; \Box_{x:\Phi(T)} \bullet \Phi\left(AA(x)\right)$$

Where the action is limited to the prefixed action AA in Φ Rule 27. □

Iterated Internal Choice

Φ Rule 37 (Iterated Internal Choice).

$$\Phi\left(\sqcap x : T \bullet A(x)\right) \quad = \quad \sqcap_{x:\Phi(T)} \bullet \Phi(A(x))$$

□

Iterated Parallel Composition The iterated parallel composition of actions is not supported yet.

Iterated Interleaving The iterated interleaving of actions is not supported yet.

4.5.10.3 Command

Variable Block

Φ Rule 38 (Variable Block). Variable block is transformed to replicated internal choice in CSP$_M$ to declare local variables x, and its body is a memory model of process $\Phi(A)$ defined in Definition B.1.5.

$$\Phi\left(\mathbf{var}\, x : T \bullet A\right) \quad = \quad \sqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi(A), \{x\}\right)$$

□

4.6 Final Considerations

4.6.1 Parallel Composition and Interleaving of Actions

Parallel composition and interleaving of actions in *Circus* are different from CSP in its two state partitions as stated in Section 2.2.1. In Φ Rule 32, the general rule is not supported. Actually, we have a solution. Our solution is to declare two sets of temporary variables tpv_1 and tpv_2 which are initialized to the values of all corresponding variables in scope pv_1 and pv_2 for A_1 and A_2. Instead of updating pv_1 and pv_2 in A_1 and A_2, we update tpv_1 and tpv_2. Eventually, only variables in ns_1 and ns_2 are updated to the values of corresponding variables in tpv_1 and tpv_2, and others are discarded.

$$\begin{array}{l} R_{wrt}\left(A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2\right) \\ = \left(\begin{array}{l} \left(\mathbf{var}\, tpv_1 \bullet \left(\begin{array}{l} tpv_1 := pv_1; \\ (A_1[tpv_1/pv_1])\,; \\ ns_1 := tpns_1 \end{array}\right)\right) \\ \llbracket ns_1 \mid cs \mid ns_2 \rrbracket \\ \left(\mathbf{var}\, tpv_2 \bullet \left(\begin{array}{l} tpv_2 := pv_2; \\ (A_2[tpv_2/pv_2])\,; \\ ns_2 := tpns_2 \end{array}\right)\right) \end{array}\right) \end{array}$$

However, it is difficult to reason about the solution and even more difficult to implement it in the translator (but easy for manual translation). Finally, we exclude this general solution in our current work.

4.6.2 Tractability of Model Checking by linking to the combination of CSP and B

Generally, a model in *Circus* is composed of processes in the similar way to that of CSP, in that the nodes in their transition systems do not have a visible state. In other words, state variables are local to processes in *Circus*. Furthermore, for an individual process, its transition relation is defined in terms of the transition relation of its main action [75, Table 1].

Comparatively, our approach to link a *Circus* model to a model in $CSP \parallel B$ may result in a bigger state space than its original state space in *Circus*. That can be seen from the following perspectives: the state part in Z or B, the behavioural part in CSP, and the combined state in $CSP \parallel B$. Since state variables in *Circus* are local to processes and hidden from other processes, internally a process can be regarded as a local transition system. For a composition of processes, only the transition relations in the main action of processes, instead of local state variables, are used to form a new transition system for composed processes. However, on the one hand, in the linked counterpart in $CSP \parallel B$, state variables from different processes are merged in the B machine to form a new big set of state variables. Therefore, from the state aspect, the model checker of $CSP \parallel B$ is harder to explore the state space than that of *Circus*. For example, if a *Circus* model has two processes, one with m state variables and another with n state variables, the model checker of $CSP \parallel B$ needs to explore a state space with $m + n$ variables. But the model checker of *Circus* only needs to explore a state space with m variables and a state space with n variables separately. On the other hand, from the behavioural part, the CSP specification in $CSP \parallel B$ also has a bigger state space than that in *Circus*. The main reason is due to the requirements of additional communication for interaction between CSP and B, such as operational events and additional events which are required in CSP to retrieve the values of state variables from B. Link Rule 35 is an example that illustrates additional events $Op\_s_i, \cdots, Op\_s_j$ are necessary. Finally, the state space of $CSP \parallel B$, which is a combination of the state space in B and the state space in CSP, is bigger than that of *Circus* in which only behavioural part is explored globally and the state part is explored locally. In sum, our approach may result in the difficulty to explore the state space of systems.

4.6.3 *Circus* versus Plain CSP and Z

As stated in Section 4.6.2, our approach to linking *Circus* to $CSP \parallel B$ (or $CSP \parallel_B Z$ in terms of $CSP \parallel B$) may cause a bigger state space and eventually harder to be checked. Then there is another question arising from this perspective. Is this difficulty due to our automatic translation approach or just because of $CSP \parallel B$? Is it possible to use a delicately designed plain $CSP \parallel_B Z$ model (here we mean the modelling by $CSP \parallel_B Z$ directly) to achieve a state space with the similar size to that of *Circus*? To answer this question, we need to analyse it from the specification style first and then from impact of the state space on performance.

Circus and $CSP \parallel_B Z$ use different styles in their specifications. *Circus* has modular and hierarchical structure while $CSP \parallel_B Z$ is flat. In a *Circus* model, the state part is local to processes and not visible from outside of individual processes. Only the behavioural part of processes in their main action is visible. By this way, information hiding is achieved. In addition, this local state part and global behaviour part facilitate compositionality and modularity like object oriented design. However $CSP \parallel B$ in ProB uses a different style. In $CSP \parallel B$, the state part is in a B machine and the behavioural part is given in CSP. Basically, it does not support encapsulation. All state variables are mixed in a same machine and they can be accessed by all CSP processes. Furthermore, *Circus* has another advantage over $CSP \parallel_B Z$ in its refinement calculus.

Compared to the resultant $CSP \parallel_B Z$ by our link rules, a well designed model by plain $CSP \parallel_B Z$ may have less additional events, for retrieving the values of state variables, than our linked $CSP \parallel_B Z$. However, fundamentally they are the same language. As analysed in Section 4.6.2, a system modelled in *Circus* has a smaller state space than the same system modelled in $CSP \parallel_B Z$. This is because the $CSP \parallel_B Z$ model has larger state spaces in terms of the state part, the behavioural part (due to the fact that more communication is required between CSP and Z), and the combination of them. Thus inevitably the performance of model checking in $CSP \parallel_B Z$ will not be as good as that of model checking in *Circus* directly.

4.7 Summary

This chapter defines the link from *Circus* to $CSP \parallel B$, namely Υ. In the first section of this chapter, the link strategies and function decompositions are present. Υ is composed of a R_{wrt} function, a Ω function, and a Φ function. The R_{wrt} function transforms the original *Circus* model into an intermediate model in which all interactions between the state part and the behavioural part are only through schema expressions. Then the state part of this rewritten model is translated to B by the Ω function which is decomposed into three sub-functions: Ω_1, Ω_2, and Ω_3. Particularly, our Ω_3 function relies on the translator in ProB to translate a Z specification in ZRM to a B machine. The behavioural part of the rewritten model is translated to CSP by the Φ function.

Then in the rest of the chapter, we give individual rules for each function. These rules can be composed together to link a construct in *Circus* to a construct in $CSP \parallel_B Z$. This is illustrated in Appendix E. In this part, not all constructs in *Circus* are supported and they are listed below.

Identifier identifiers are limited to the pattern below `[a-zA-Z][a-zA-Z0-9_]['!?]` as shown in Section 4.2.1

External Choice of Processes both processes should be prefixed processes (R_{wrt} Rule 22)

External Choice of Actions both actions should be prefixed actions (R_{wrt} Rule 30) or mutually exclusive guarded actions (Φ Rule 28)

Recursion recursion is partially restricted to be an action of external choice (R_{wrt} Rule 34)

Parallel Composition and Interleaving of Actions only "Disjoint Variables in Scope" (Φ Rule 30) and "Disjoint Variables in Updating" (Φ Rule 31) are supported and the general one is not supported (Φ Rule 32)

Chapter 5

Soundness

This chapter demonstrates the soundness of our defined link from *Circus* to $CSP \parallel_B Z$. In the beginning of this chapter, the strategies of how to prove soundness is provided in Section 5.1. Then two important theorems about the semantics of a typical construct in the combination of CSP and Z in terms of $CSP \parallel B$ are presented and proved in Section 5.2. They facilitate the soundness proof of the link for schema expression as action in Section 5.10.1. Finally, the soundness for each individual link rule is illustrated in the remaining sections.

5.1 Strategies

5.1.1 Overall Strategies

Our overall strategies to prove the soundness of the defined link from *Circus* to $CSP \parallel_B Z$ is given in Theorem 5.1.1.

Theorem 5.1.1. If the link rules, defined in Chapter 4 and summarised in Appendix E, are sound for all constructs given, then the link from *Circus* to $CSP \parallel_B Z$ is sound.

Proof. To prove the soundness of the link from *Circus* to $CSP \parallel_B Z$, it is difficult to reason about the semantics of *Circus* and $CSP \parallel_B Z$ programs as a whole simply. In other words, we cannot give UTP semantics to a *Circus* model and a $CSP \parallel_B Z$ model and then conclude their semantics are equal. Our solution is to treat a *Circus* model as a combination of a collection of constructs defined in our link. Then in order to prove the link from *Circus* to $CSP \parallel_B Z$ is sound, our strategy is to prove our link rules given in Appendix E are sound for each construct. Finally, if the link rules for all constructs defined are sound, then we can conclude the link from *Circus* to $CSP \parallel_B Z$ is sound.

The soundness of link rules for individual construct is given in sections from Section 5.3 to Section 5.10 and in lemmas from Lemma 5.3.1 to Lemma 5.10.38. □

5.1.2 Individual Strategies

In order to prove the soundness of each individual link rule for a construct, we use UTP to establish the relation between a construct in *Circus* and the linked counterpart in $CSP \parallel_B Z$ because the semantics of *Circus* is given in UTP. The denotational semantics of *Circus* is given in Oliveira's PhD thesis [35] and the paper [73].

Furthermore, for the constructs that specify the state part in *Circus*—actually it is Z in ISO Standard dialect, their semantics are defined in the design theory of UTP. While for the constructs that specify the behavioural part in *Circus*, their semantics are given in the reactive design theory of UTP.

5.1.2.1 Steps to Prove

The proof of the soundness of an individual link rule is composed of several steps as follows.

Step 1 at first, the semantics of the original construct in *Circus* is given according to Oliveira's thesis [35] and the paper [73].

Step 2 the semantics of the resultant construct in $CSP \parallel_B Z$, which is obtained by the application of link rules in Appendix E to the original construct, is given.

Step 3 a comparison of the semantics of the original construct in *Circus*, given in **Step 1**, and the semantics of the resultant construct in $CSP \parallel_B Z$, given in **Step 2**, is undertaken. If they have the same semantics, therefore the link rule defined for the syntax in *Circus* is sound.

Step 4 Otherwise, if their semantics are not equal, we modify the link rule to partially support the construct by adding limitations or completely exclude it from our link definition.

In addition, with respect to each individual link rule, its proof is based on the assumption that the link rules for all other constructs is sound. For instance, Link Rule 39 links a *Circus* construct

$$(g) \mathbin{\&} A$$

to the construct in $CSP \parallel_B Z$

$$\Phi\left(R_{mrg}\left(R_{pre}\left(g\right), R_{pre}\left(A\right)\right)\right) \rightarrow\ \Phi\left(g\right)\ \mathbin{\&} \Phi\left(R_{post}\left(A\right)\right)$$

When proving the soundness of this link rule, we assume $\Phi\left(g\right)$ and $\Phi\left(R_{wrt}\left(A\right)\right)$ are sound links of g and A in *Circus* to $CSP \parallel_B Z$. Particularly, the soundness of link rules for $\Phi\left(g\right)$ and $\Phi\left(R_{wrt}\left(A\right)\right)$ will be proved in their own proofs and not in the proof of this guarded action link rule.

5.2 UTP Semantics of the Combination of CSP and Z Programs

Two theorems below give the semantics to a typical $CSP \parallel_B Z$ construct that is the only way of interaction between the state part in Z and the behavioural part in CSP through operations. These theorems characterise the interaction. Therefore, they are important in $CSP \parallel_B Z$. In addition, actually in our link definition, this typical construct in $CSP \parallel_B Z$ is a counterpart of schema expression as action in *Circus*. Since our rewrite rules transform all interactions between the state part and the behavioural part in *Circus* into schema expressions as action. Sequentially they will be linked to the typical construct in $CSP \parallel_B Z$. Therefore, when we reason about the semantics of constructs like specification statement and assignment in *Circus*, we also need these theorems because specification statement and assignment are rewritten to schema expressions. Finally, we put these theorems in early section of this chapter.

5.2.1 Theorem 1

Theorem 5.2.1. The UTP semantics of the $CSP \parallel_B Z$ program below,

$$\left.\begin{array}{l} channel\ SExp : \Phi\left(T_i\right).\Phi\left(T_o\right) \\ channel\ SExp\_fOp : \Phi\left(T_i\right) \\ HIDE\_CSPB = \{\!\mid P\_SExp, P\_SExp\_fOp, \cdots \mid\!\} \\ P\_SExp!l_o?l_i \rightarrow SKIP \mathbin{\square} P\_SExp\_fOp!l_o \rightarrow \mathbf{div} \end{array}\right\} CSP$$

$$\left.\begin{array}{l}
P\_SExp == [\, decl\, ; \Xi Q_1\_StPar\, ; \cdots \Xi Q_n\_StPar;\\
\qquad ins? : T_i\, ; outs! : T_o \mid pred\,]\\
P\_SExp\_fOp == [\, \Xi P\_StPar\, ; \Xi Q_1\_StPar\, ; \cdots \Xi Q_n\_StPar;\\
\qquad ins? : T_i \mid \neg\, \mathbf{pre}\, P\_SExp]
\end{array}\right\} Z$$

is

$$\boldsymbol{R}\left(pre\, [l_o/ins?] \vdash post\, [l_o, l_i/ins?, outs!] \wedge \neg wait' \wedge tr' = tr\right)$$

where

- ins and $outs$ denotes input and output variables of the P_SExp schema, and T_i and T_o are their corresponding types in *Circus*.
- Correspondingly, l_o and l_i denotes output and input local variables on the same name channel P_SExp in CSP.
- pre and $post$ are the precondition and postcondition of the P_SExp operation, and $pre \wedge post = pred$. Since P_SExp is an operation, it shall include all dashed state variables. Finally, $post$ in the postcondition of $\boldsymbol{R}$ actually characterises the state part of $CSP \parallel_B Z$.
- $Q_1, \cdots, Q_n$ are all processes other than P.

It states that if the precondition of P_SExp holds (pre is $true$), it terminates successfully leaving the trace unchanged and the postcondition established. In addition, for the variables not in the frame, they remain unchanged. Conversely, if the precondition of P_SExp does not hold, it diverges.

Proof. (1). According to $CSP \parallel B$ model 2.2, this $CSP \parallel_B Z$ program is modelled as below.

$$\left(\begin{array}{l}
\left(\mu X \bullet \left(\begin{array}{l}
((P\_SExp?ins!outs \to X) \lhd enabled\, (P\_SExp) \rhd STOP)\\
\Box\\
((P\_SExp\_fOp?ins \to X) \lhd enabled\, (P\_SExp\_fOp) \rhd STOP)\\
\Box\, (((op_1 \to X) \lhd enabled\, (Op_1) \rhd STOP)\\
\Box \ldots\\
\Box\, ((op_n \to X) \lhd enabled\, (Op_n) \rhd STOP) \vdots
\end{array}\right)\right)_{v}^{v'}\\
\underset{\{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\}}{{}_B\|_C}\\
(P\_SExp!l_o?l_i \to SKIP \,\Box\, P\_SExp\_fOp!l_o \to \mathbf{div})
\end{array}\right)\\
\setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\}$$

Provided $enabled\, (P\_SExp[l_o/ins?]) = true$, then $enabled\, (P\_SExp\_fOp[l_o/ins?]) = false$. Hence, the equation above is simplified to

$$\left(\begin{array}{l}
\left(\mu X \bullet \left(\begin{array}{l}
(P\_SExp?ins!outs \to X)\\
\Box\\
(STOP)\\
\vdots
\end{array}\right)\right)_{v}^{v'}\\
\underset{\{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\}}{{}_B\|_C}\\
(P\_SExp!l_o?l_i \to SKIP \,\Box\, P\_SExp\_fOp!l_o \to \mathbf{div})
\end{array}\right)\\
\setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\}$$

$$[enabled\, (P\_SExp) = true]$$

$$= \left\{ \begin{array}{l} \exists\, v_0 \bullet \\ \left(\begin{array}{l} P\_SExp!l_o?l_i \rightarrow \\ \left(\begin{array}{l} \left(\left(\mu\, X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (P\_SExp?ins!outs \rightarrow X) \\ \qquad \lhd enabled\,(P\_SExp) \rhd STOP \end{array} \right) \\ \Box \\ \left(\begin{array}{l} (P\_SExp\_fOp?ins \rightarrow X) \\ \qquad \lhd enabled\,(P\_SExp\_fOp) \rhd STOP \end{array} \right) \\ \vdots \end{array} \right) \right) \right)^{v'}_{v_0} \\ \underset{\{\!|P\_SExp, P\_SExp\_fOp, \cdots|\!\}}{{}_B\|_C} \\ (SKIP) \end{array} \right) \end{array} \right) \\ \setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\} \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i, v_0 / ins?, outs!, v'] \end{array} \right.$$

[Law 2.4.3]

$$= \left\{ \begin{array}{l} \exists\, v_0 \bullet \\ (P\_SExp!l_o?l_i \rightarrow SKIP) \setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\} \\ \land_R\ (v' = v_0) \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i, v_0 / ins?, outs!, v'] \end{array} \right. \qquad \text{[Law 2.4.1]}$$

$$= \left\{ \begin{array}{l} \exists\, v_0 \bullet \\ SKIP \setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\} \\ \land_R\ (v' = v_0) \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i, v_0 / ins?, outs!, v'] \end{array} \right. \qquad \text{[Hiding [7, Section 3.5.1, L5]]}$$

$$= \left\{ \begin{array}{l} \exists\, v_0 \bullet \\ SKIP \\ \land_R\ (v' = v_0) \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i, v_0 / ins?, outs!, v'] \end{array} \right. \qquad \text{[Hiding [7, Section 3.5.1, L5]]}$$

$$= \left\{ \begin{array}{l} SKIP \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i, v_0 / ins?, outs!, v'][v'/v_0] \end{array} \right.$$

[Predicate Calculus and ***R3***- $\land$ -closure [68]]

$$= \left\{ \begin{array}{l} SKIP \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i, v' / ins?, outs!, v'] \end{array} \right. \qquad \text{[Substitution]}$$

$$= \left\{ \begin{array}{l} SKIP \\ \land_R\ (post\,(P\_SExp))\,[l_o, l_i / ins?, outs!] \end{array} \right.$$

The result shows that this program terminates successfully as the behaviour is the same as $SKIP$ and the postcondition of P_SExp is established from the state perspective. At the same time, the variables not in the frame remain unchanged.

However, if $enabled\,(P\_SExp[l_o/ins?]) = false$, then $enabled\,(P\_SExp\_fOp[l_o/ins?]) = true$. As a result, the equation can be simplified to

$$\left(\begin{array}{l} \left(\mu\, X \bullet \left(\begin{array}{l} (STOP) \\ \Box \\ (P\_SExp\_fOp?ins \rightarrow X) \\ \vdots \end{array} \right) \right)^{v'}_{v} \\ \underset{\{\!|P\_SExp, P\_SExp\_fOp, \cdots|\!\}}{{}_B\|_C} \\ (P\_SExp!l_o?l_i \rightarrow SKIP \;\Box\; P\_SExp\_fOp!l_o \rightarrow \mathbf{div}) \end{array} \right) \\ \setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\}$$

$[enabled\,(P\_SExp) = false]$

$$
= \begin{array}{l}
\exists\, v_0 \bullet \\
\left(\begin{array}{l}
P\_SExp\_fOp!l_o \rightarrow \\
\left(\begin{array}{l}
\left(\mu\, X \bullet \left(\begin{array}{l}
\left(\begin{array}{l} (P\_SExp?ins!outs \rightarrow X) \\ \quad \lhd enabled\,(P\_SExp) \rhd STOP \end{array} \right) \\
\Box \\
\left(\begin{array}{l} (P\_SExp\_fOp?ins \rightarrow X) \\ \quad \lhd enabled\,(P\_SExp\_fOp) \rhd STOP \end{array} \right) \\
\vdots
\end{array} \right) \right)_{v_0}^{v'} \\
\quad {}_{B}\|_{C} \\
\{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\} \\
(\mathbf{div})
\end{array} \right)
\end{array} \right) \\
\setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\} \\
\wedge_R\; post\,(P\_SExp\_fOp)\,[v_0/v']
\end{array}
$$

[Law 2.4.3]

$$
= \left\{ \begin{array}{l}
\exists\, v_0 \bullet \\
(P\_SExp\_fOp?ins \rightarrow \mathbf{div}) \setminus \{\!| P\_SExp, P\_SExp\_fOp, \cdots |\!\} \\
\wedge_R\; \mathbf{true} \\
\wedge_R\; post\,(P\_SExp\_fOp)\,[v_0/v']
\end{array} \right. \qquad \text{[Law 2.4.1]}
$$

$$
= \left\{ \begin{array}{l}
\exists\, v_0 \bullet \\
(\mathbf{div}) \\
\wedge_R\; \mathbf{true} \\
\wedge_R\; (v_0 = v)
\end{array} \right.
$$

[Hiding [7, Section 3.5.1, L5] and postcondition of P_SExp_fOp is $true$]

$$
= \left\{ \begin{array}{l}
(\mathbf{div}) \\
\wedge_R\; \mathbf{true}[v/v_0]
\end{array} \right. \qquad \text{[Predicate Calculus and } \boldsymbol{R3}\text{-} \wedge \text{-closure [68]]}
$$

$$
= \left\{ \begin{array}{l}
(\mathbf{div}) \\
\wedge_R\; \mathbf{true}
\end{array} \right. \qquad \text{[Substitution]}
$$

$$
= \ (\mathbf{div}) \qquad \text{[Predicate Calculus]}
$$

Since the predicate of P_SExp_fOp only includes the negation of the precondition of P_SExp, it will not change the state and $v' = v$. But after that, the program diverges and makes v' unconstrained.

(2). The UTP semantics of this $CSP \parallel_B Z$ program is given as

$$\boldsymbol{R}\left(pre[l_o/ins?] \vdash post[l_o, l_i/ins?, outs!] \wedge \neg wait' \wedge tr' = tr\right)$$

This semantics states that if $pre[l_o/ins?]$ is $true$ (the precondition of P_SExp holds when its input variables $ins?$ get the value from the local variables l_o in CSP, or the P_SExp operation is $enabled$), it terminates successfully leaving the trace unchanged and the post-condition of P_SExp, $post[l_o, l_i/ins?, outs!]$, is established. Otherwise, if $pre[l_o/ins?]$ is $false$, it diverges.

This statement is exactly the same as the behaviour and the state for both situations in (1). Accordingly, this semantics for the program is proved. □

5.2.2 Theorem 2

Theorem 5.2.2. The UTP semantics of the $CSP \parallel_B Z$ program below

$$
\left. \begin{array}{l}
\text{channel P\_SExp:}\Phi\,(T_i)\,.\Phi\,(T_o) \\
\text{HIDE\_CSPB=}\{\!| P\_SExp, \cdots |\!\} \\
P\_SExp!l_o?l_i \rightarrow SKIP
\end{array} \right\} CSP
$$

$$
\left. \begin{array}{l}
P\_SExp == [\, decl\,;\, \Xi Q_1\_StPar\,;\, \cdots \Xi Q_n\_StPar; \\
\qquad\qquad ins? : T_i\,;\, outs! : T_o \mid pred\,]
\end{array} \right\} Z
$$

is

$$\mathbf{R}\left(\begin{array}{l} (pre[l_o/ins?] \Rightarrow true) \\ \vdash \\ \left(\begin{array}{l} (pre[l_o/ins?] \wedge post[l_o, l_i/ins?, outs!] \wedge \neg wait' \wedge tr' = tr) \\ \vee \\ (\neg pre[l_o/ins?] \wedge tr' = tr \wedge wait') \end{array}\right) \end{array}\right)$$

It states that if the precondition of P_SExp holds, it terminates successfully leaving trace unchanged and the postcondition is established. Conversely, if the precondition of P_SExp does not hold, it blocks like $STOP$ as

$$STOP \mathrel{\widehat{=}} \mathbf{R}\left(true \vdash tr' = tr \wedge wait'\right) \qquad \text{[Cavalcanti \& Woodcock [68, Section 6.3]]}$$

Proof. (1). According to $CSP \parallel B$ model 2.2, this $CSP \parallel_B Z$ program is modelled as below.

$$\left(\begin{array}{l} \left(\mu X \bullet \left(\begin{array}{l} ((P\_SExp?ins!outs \rightarrow X) \lhd enabled\,(P\_SExp) \rhd STOP) \\ \Box \\ \Box\ (((op_1 \rightarrow X) \lhd enabled\,(Op_1) \rhd STOP) \\ \Box \ldots \\ \Box\ ((op_n \rightarrow X) \lhd enabled\,(Op_n) \rhd STOP) \end{array}\right)\right)_v^{v'} \\ {}_B\|_C \\ \{\!|P\_SExp, \cdots|\!\} \\ (P\_SExp!l_o?l_i \rightarrow SKIP) \end{array}\right) \\ \setminus \{\!|P\_SExp, \cdots|\!\}$$

Provided $enabled\,(P\_SExp[l_o/ins?]) = true$, the equation above can be simplified to

$$\left(\begin{array}{l} \left(\mu X \bullet \left(\begin{array}{l} (P\_SExp?ins!outs \rightarrow X) \\ \Box \\ \vdots \end{array}\right)\right)_v^{v'} \\ {}_B\|_C \\ \{\!|P\_SExp, \cdots|\!\} \\ (P\_SExp!l_o?l_i \rightarrow SKIP) \end{array}\right) \setminus \{\!|P\_SExp, \cdots|\!\}$$

[$enabled\,(P\_SExp) = true$]

$$= \left\{\begin{array}{l} \exists v_0 \bullet \\ \left(\begin{array}{l} P\_SExp!l_o?l_i \rightarrow \\ \left(\begin{array}{l} \left(\mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (P\_SExp?ins!outs \rightarrow X) \\ \qquad \lhd enabled\,(P\_SExp) \rhd STOP \end{array}\right) \\ \Box \\ \vdots \end{array}\right)\right)_{v_0}^{v'} \\ {}_B\|_C \\ \{\!|P\_SExp, \cdots|\!\} \\ (SKIP) \end{array}\right) \end{array}\right) \\ \setminus \{\!|P\_SExp, \cdots|\!\} \\ \wedge_R\ (post\,(P\_SExp))\,[l_o, l_i, v_0/ins?, outs!, v'] \end{array}\right.$$

[Law 2.4.3]

$$= \left\{\begin{array}{l} \exists v_0 \bullet \\ (P\_SExp!l_o?l_i \rightarrow SKIP) \setminus \{\!|P\_SExp, \cdots|\!\} \\ \wedge_R\ (v' = v_0) \\ \wedge_R\ post\,(P\_SExp)\,[l_o, l_i, v_0/ins?, outs!, v'] \end{array}\right. \qquad \text{[Law 2.4.1]}$$

$$= \left\{ \begin{array}{l} \exists\, v_0 \bullet \\ (SKIP) \\ \wedge_R \, (v' = v_0) \\ \wedge_R \, post\,(P\_SExp)\,[l_o, l_i, v_0/ins?, outs!, v'] \end{array} \right. \qquad \text{[Hiding [7, Section 3.5.1, L5]]}$$

$$= \left\{ \begin{array}{l} (SKIP) \\ \wedge_R \, (post\,(P\_SExp))\,[l_o, l_i, v'/ins?, outs!, v'] \end{array} \right.$$

[Predicate Calculus and Substitution]

$$= \left\{ \begin{array}{l} (SKIP) \\ \wedge_R \, (post\,(P\_SExp))\,[l_o, l_i/ins?, outs!] \end{array} \right.$$

Provided $enabled\,(P\_SExp[l_o/ins?]) = false$, the equation can be simplified to

$$\left(\begin{array}{l} \left(\mu\, X \bullet \left(\begin{array}{l} (STOP) \\ \Box \\ \vdots \end{array} \right) \right)_v^{v'} \\ \quad {}_B\|_C \\ \{\!|P\_SExp, \cdots|\!\} \\ (P\_SExp!ins?outs \rightarrow SKIP) \end{array} \right) \setminus \{\!|P\_SExp, \cdots|\!\}$$

$[enabled\,(P\_SExp) = false]$

$$= STOP \wedge_R \big(v' = v\big) \qquad \text{[Law 2.4.3]}$$

Finally, if P_SExp is not enabled, the program deadlocks and no state will be changed.

(2). The UTP semantics of this $CSP \parallel_B Z$ program is given as

$$\boldsymbol{R} \left(\begin{array}{l} (pre[l_o/ins?] \Rightarrow true) \\ \vdash \\ \left(\begin{array}{l} (pre[l_o/ins?] \wedge post[l_o, l_i/ins?, outs!] \wedge \neg wait' \wedge tr' = tr) \\ \vee \\ (\neg pre[l_o/ins?] \wedge tr' = tr \wedge wait') \end{array} \right) \end{array} \right)$$

This semantics states that if $pre[l_o/ins?]$ is $true$ (the precondition of P_SExp holds when its input variables $ins?$ get the value from the local variables l_o in CSP, or the P_SExp operation is *enabled*), it terminates successfully leaving the trace unchanged and the post-condition of P_SExp, $post[l_o, l_i/ins?, outs!]$, is established. Otherwise, if $pre[l_o/ins?]$ is $false$, it deadlocks.

This statement is exactly the same as the behaviour and the state for both situations in (1). Accordingly, this semantics for the program is proved. □

5.3 Channel Declaration

From this section, we will prove the soundness of a link rule (given in Appendix E) for each construct that is defined in the link.

5.3.1 Synchronisation Channel

Lemma 5.3.1 (Synchronisation Channel). *Link Rule 1 for synchronisation channel declaration is sound.*

Proof. A synchronisation channel declaration in *Circus* is linked to a synchronisation channel declaration in CSP as well according to Link Rule 1. It declares a set of channels by their name but without type in *Circus* which is the same as that in CSP. □

5.3.2 Typed Channel

Lemma 5.3.2 (Typed Channel)**.** *Link Rule 2 for typed channel declaration is sound.*

Proof. A typed channel declaration in *Circus* is linked to a typed channel declaration in CSP as well according to Link Rule 2. Both of them declare a set of channels by their name and type but the type T in *Circus* is linked to $\Phi(T)$ in CSP. According to Φ Rule 1, it is the same representation of T. □

5.3.3 Schema Typed Channel

Lemma 5.3.3 (Schema Typed Channel)**.** *Link Rule 3 for schema typed channel declaration is sound.*

Proof. A schema typed channel declaration itself is just a syntactical group of typed channel declarations. It is linked to a set of typed channel declarations by Link Rule 3. □

5.4 Channel Set Declaration

Lemma 5.4.1 (Channel Set Declaration)**.** *Link Rule 4 for channel set declaration is sound.*

Proof. A channel set paragraph associates a channel set name N to a channel set expression $CSExp$, which is the same as the linked construct

$$N = \Phi\left(CSExp\right)$$

in CSP by Link Rule 4. Additionally, $\Phi\left(CSExp\right)$ is a sound link of $CSExp$ according to Section 5.5. □

5.5 Channel Set Expressions

Lemma 5.5.1 (Channel Set Expressions)**.** *Link Rule 5, 6, and 7 for channel set expressions are sound.*

Proof. An empty channel set in *Circus* is linked to an empty channel set in CSP by Link Rule 5.

A channel enumeration expression declares a set of all events associated with channels inside and its semantics is given by *extensions* and *productions* closure operations [58]. The $\{|\,|\}$ operator in CSP is defined by *productions* operation as well. Therefore Link Rule 6 preserves the semantics.

A reference to a channel set expression is the name of the channel set by Link Rule 7.

And channel set expressions by set union $\cup$, set intersection $\cap$, and set difference $\setminus$ are linked by Link Rule 7 to corresponding operators union, inter, and diff in CSP. Since these operators are directly mapped from the counterparts in *Circus* and the map is given in Table D.5, the channel set expressions in CSP have the same semantics as those in *Circus*. □

5.6 Explicitly Defined Processes

5.6.1 Single Explicitly Defined Process

Lemma 5.6.1 (Single Explicitly Defined Process)**.** *Link Rule 8 for a single explicitly defined process is sound.*

Proof. (1). An explicitly defined process P in *Circus* declares state components $(s_1, \cdots, s_n)$ by *Circus* variable block and behaves like its main action A.

$$\left(\begin{array}{l}\textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\ \qquad \textbf{state}\ StPar == [\, s_1 : T_1\, ; \cdots s_n : T_n \mid pred\,] \\ \qquad Pars == [\cdots] \\ \qquad \bullet\ A \\ \textbf{end}\end{array}\right)$$

$$\mathrel{\widehat=} \textbf{var}\ s_1 : T_1\, ; \cdots s_n : T_n \bullet A \qquad \text{[Oliveira [35, Definition B.41]]}$$

(2). An explicitly defined process P is linked to a $CSP \parallel_B Z$ program by Link Rule 8. The *State* schema in Z declares a set of state components which is the same as that in *Circus* (though each state component is renamed), and its behaviour is given by the CSP process, that is, $P = \Phi\left(R_{wrt}(A)\right)$. According to our soundness proof strategies, $\Phi\left(R_{wrt}(A)\right)$ is a semantically equal link of A in *Circus* to CSP. Therefore, the linked $CSP \parallel_B Z$ program has the same semantics as the explicitly defined process in *Circus* and the rule is sound.

□

5.6.2 Multiple Explicitly Defined Processes

Lemma 5.6.2 (Multiple Explicitly Defined Processes)**.** *Link Rule 9 for multiple explicitly defined processes is sound.*

Proof. (1). According to Definition B.42 to Definition B.45 in Oliveira's thesis [35], for the composition of multiple explicitly defined processes, the final state is a conjunction of states from all these processes, and each schema from one process, which specifies an operation on its own state, is lifted to an operation on the global state by conjoining with all other processes' state paragraphs ($\Xi P_i.State$). And the composition operator on multiple processes accordingly becomes the corresponding action operator on the main actions of these processes. For instance,

$$P\ op^p\ Q \mathrel{\widehat=} \left(\begin{array}{l}\textbf{begin}\,\textbf{state}\ State == P.StPar \wedge Q.StPar \\ \qquad P.Pars \wedge_{\Xi} Q.StPar \\ \qquad Q.Pars \wedge_{\Xi} P.StPar \\ \qquad \bullet\ P.A\ op^a\ Q.A \\ \textbf{end}\end{array}\right)$$

[Oliveira [35, Definition B.42 to B.45]]

where $P.Pars \wedge_{\Xi} Q.StPar$ means for each schema Par in P to conjunct with $\Xi Q.StPar$ which further indicates the state components in Q will not be updated in $P.Pars$. op^p is the composition operator on processes and op^a is the corresponding operator on actions.

(2). Explicitly defined processes are linked to a $CSP \parallel_B Z$ program by Link Rule 9. The final state

$$State \mathrel{\widehat=} P_1\_StPar \wedge \cdots \wedge P_n\_StPar$$

is a conjunction of states from all processes. And for each schema from one process

$$Pn\_Pars \mathrel{\widehat=} [P_n\_Pars.decl_n\, ; \Xi P_1\_StPar\, ; \ldots ; \Xi P_{n-1}\_StPar \mid P_n\_Pars.p_n]$$

it is transformed to a new schema operating on the global state and the new schema includes the original declarations and predicate as well as all other processes' decorated state paragraphs, ΞP_i_StPar, in its declaration part. Furthermore, the main action of each process P_i has been linked to a same name process, $P_i = \Phi\left(R_{wrt}\left(P_i.A\right)\right)$, in CSP. However, for the process operator and its corresponding action operator, they are not linked in this rule. They are linked in the link rule of each individual process operator, such as Link Rule 10.

(3). Finally, we can conclude that the link rule for explicitly defined processes to merge states and schemas and link the main actions actually results in the same semantics construct in $CSP \parallel_B Z$ as that in *Circus*, and the link is sound. □

5.7 Compound Processes

5.7.1 Sequential Composition

Lemma 5.7.1 (Sequential Composition of Processes)**.** *Link Rule 10 for sequential composition of processes is sound.*

Proof. (1). The sequential composition of two processes in *Circus* is defined below

$$P \mathbin{;} Q \mathrel{\widehat=} \left(\begin{array}{l} \textbf{begin}\,\textbf{state}\; State == P.StPar \land Q.StPar \\ \qquad P.Pars \land_{\Xi} Q.StPar \\ \qquad Q.Pars \land_{\Xi} P.StPar \\ \qquad \bullet\; P.A \mathbin{;} Q.A \\ \textbf{end} \end{array} \right)$$

[Oliveira [35, Definition B.42]]

where the main action of a sequential composition of two processes P and Q is a sequential composition of the main action of P and the main action of Q.

(2). A sequential composition of two processes is linked to a $CSP \parallel_B Z$ program $P \mathbin{;} Q$ by Link Rule 10.

Since the states, the schemas and the main actions of P and Q have been linked by Link Rule 9, this rule only links sequential composition. And

$$P \mathbin{;} Q = \Phi\left(R_{wrt}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)$$

because

$$P = \Phi\left(R_{wrt}\left(P.A\right)\right)$$
$$Q = \Phi\left(R_{wrt}\left(Q.A\right)\right) \qquad \text{[Link Rule 9]}$$

(3). Additionally, according to Link Rule 40,

a) provided this sequential composition of processes, $P \mathbin{;} Q$, is not a process in external choice, then

$$\begin{aligned} &\Phi\left(R_{wrt}\left(P.A \mathbin{;} Q.A\right)\right) \\ &= \Phi\left(R_{pre}\left(P.A\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) && \text{[Link Rule 40]} \\ &= \left(\Phi\left(R_{pre}\left(P.A\right)\right) \rightarrow \Phi\left(R_{post}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) \\ & && \text{[Hoare [7, Section 5.2, L3]]} \\ &= \left(\Phi\left(R_{wrt}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) && \text{[Definition 4.3.1]} \end{aligned}$$

b) provided this sequential composition of processes, $P \mathbin{;} Q$, is a process in external choice, then P has to be a PA, a prefixed process given in Definition B.4.1, according to R_{wrt} Rule 22 and $R_{pre}\left(P.A\right)$ is empty according to Definition 4.3.1 and Definition B.4.1 where the first event of A does not evaluate state variables. Then

$$\begin{aligned} &\Phi\left(R_{wrt}\left(P.A \mathbin{;} Q.A\right)\right) \\ &= \Phi\left(R_{pre}\left(P.A\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) && \text{[Link Rule 40]} \\ &= \left(\Phi\left(R_{post}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) && [R_{pre}\text{ is empty}] \\ &= \left(\Phi\left(R_{wrt}\left(P.A\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) && \text{[Definition 4.3.1]} \end{aligned}$$

Thus, sequential composition ; in CSP is the linked counterpart of sequential composition ; of actions in *Circus*.

Therefore, the linked sequential composition of two processes in CSP has the same behaviour as the original sequential composition in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 10 has the same semantics as sequential composition in *Circus* and the rule is sound. □

5.7.2 External Choice

Lemma 5.7.2 (External Choice of Processes)**.** *Link Rule 11 for external choice of processes is sound.*

Proof. (1). The external choice of two prefixed processes (PA in Definition B.4.1) in *Circus* is defined below

$$PA_1 \Box PA_2 \mathrel{\widehat=} \left(\begin{array}{l} \mathbf{begin\,state}\; State == PA_1.StPar \land PA_2.StPar \\ \quad PA_1.Pars \land_{\Xi} PA_1.StPar \\ \quad PA_2.Pars \land_{\Xi} PA_2.StPar \\ \quad \bullet\; PA_1.A \Box PA_2.A \\ \mathbf{end} \end{array}\right)$$

[Oliveira [35, Definition B.42]]

where the main action of external choice of two prefixed processes is an external choice of the main action of PA_1 and the main action of PA_2.

(2). The external choice of two processes is linked to a $CSP \parallel_B Z$ program $PA_1 \Box PA_2$ by Link Rule 11.

Since the states, the schemas and the main actions of PA_1 and PA_2 have been linked by Link Rule 9, this rule only links external choice. And

$$PA_1 \Box PA_2 = \Phi\left(R_{wrt}\left(PA_1.A\right)\right) \Box \Phi\left(R_{wrt}\left(PA_2.A\right)\right)$$

because

$$\begin{aligned} PA_1 &= \Phi\left(R_{wrt}\left(PA_1.A\right)\right) \\ PA_2 &= \Phi\left(R_{wrt}\left(PA_2.A\right)\right) && \text{[Link Rule 9]} \end{aligned}$$

(3). Additionally, according to Link Rule 41,

$$\begin{aligned} &\Phi\left(R_{wrt}\left(PA_1.A \Box PA_2.A\right)\right) \\ &= \Phi\left(R_{mrg}\left(R_{pre}\left(PA_1.A\right), R_{pre}\left(PA_2.A\right)\right)\right) \to \left(\Phi\left(R_{post}\left(PA_1.A\right)\right) \Box \Phi\left(R_{post}\left(PA_2.A\right)\right)\right) \\ &\qquad\qquad \text{[Link Rule 41]} \\ &= \left(\Phi\left(R_{post}\left(PA_1.A\right)\right) \Box \Phi\left(R_{post}\left(PA_2.A\right)\right)\right) \\ &\qquad\qquad [R_{pre}\left(PA_1.A\right) \text{ and } R_{pre}\left(PA_2.A\right) \text{ are empty, see Definition B.4.1}] \\ &= \left(\Phi\left(R_{wrt}\left(PA_1.A\right)\right) \Box \Phi\left(R_{wrt}\left(PA_2.A\right)\right)\right) \qquad \text{[Definition 4.3.1]} \end{aligned}$$

thus external choice in CSP is the linked counterpart of external choice of actions in *Circus*.

Therefore, the linked external choice of two processes in CSP has the same behaviour as the original external choice in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 11 has the same semantics as external choice in *Circus* and the rule is sound. □

5.7.3 Internal Choice

Lemma 5.7.3 (Internal Choice of Processes)**.** *Link Rule 12 for internal choice of processes is sound.*

Proof. (1). The internal choice of two processes in *Circus* is defined below

$$P \sqcap Q \mathrel{\widehat=} \left(\begin{array}{l} \textbf{begin}\,\textbf{state}\, State == P.StPar \land Q.StPar \\ \quad P.Pars \land_{\Xi} Q.StPar \\ \quad Q.Pars \land_{\Xi} P.StPar \\ \quad \bullet\ P.A \sqcap Q.A \\ \textbf{end} \end{array}\right)$$

[Oliveira [35, Definition B.42]]

where the main action of internal choice of two processes is an internal choice of the main action of P and the main action of Q.

(2). The internal choice of two processes is linked to a $CSP \parallel_B Z$ program $P \sqcap Q$ by Link Rule 12.

Since the states, the schemas and the main actions of P and Q have been linked by Link Rule 9, this rule only links internal choice. And

$$P \sqcap Q = \Phi\left(R_{wrt}\left(P.A\right)\right) \sqcap \Phi\left(R_{wrt}\left(Q.A\right)\right)$$

because

$$\begin{aligned} &P = \Phi\left(R_{wrt}\left(P.A\right)\right) \\ &Q = \Phi\left(R_{wrt}\left(Q.A\right)\right) && \text{[Link Rule 9]} \end{aligned}$$

(3). Additionally, according to Link Rule 42,

a) provided this internal choice of processes, $P \sqcap Q$, is not a process in external choice, then

$$\begin{aligned} &\Phi\left(R_{wrt}\left(P.A \sqcap Q.A\right)\right) \\ &= \Phi\left(R_{mrg}\left(R_{pre}\left(P.A\right), R_{pre}\left(Q.A\right)\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(P.A\right)\right) \sqcap \Phi\left(R_{post}\left(Q.A\right)\right)\right) \\ &\qquad \text{[Link Rule 42]} \\ &= \left(\Phi\left(R_{pre}\left(P.A\right)\right) \rightarrow \Phi\left(R_{post}\left(P.A\right)\right)\right) \sqcap \left(\Phi\left(R_{pre}\left(Q.A\right)\right) \rightarrow \Phi\left(R_{post}\left(Q.A\right)\right)\right) \\ &\qquad \text{[Hoare [7, Section 3.2.1, L5], Definition 4.3.1 and Definition 4.3.2]} \\ &= \left(\Phi\left(R_{wrt}\left(P.A\right)\right) \sqcap \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) \qquad \text{[Definition 4.3.1]} \end{aligned}$$

b) provided this internal choice of processes, $P \sqcap Q$, is a process in external choice, then P and Q have to be PA according to R_{wrt} Rule 22 and both $R_{pre}\left(P.A\right)$ and $R_{pre}\left(Q.A\right)$ are empty because of Definition B.4.1. Then

$$\begin{aligned} &\Phi\left(R_{wrt}\left(P.A \sqcap Q.A\right)\right) \\ &= \Phi\left(R_{mrg}\left(R_{pre}\left(P.A\right), R_{pre}\left(Q.A\right)\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(P.A\right)\right) \sqcap \Phi\left(R_{post}\left(Q.A\right)\right)\right) \\ &\qquad \text{[Link Rule 42]} \\ &= \left(\Phi\left(R_{post}\left(P.A\right)\right) \sqcap \Phi\left(R_{post}\left(Q.A\right)\right)\right) \qquad \text{[Both } R_{pre} \text{ are empty]} \\ &= \left(\Phi\left(R_{wrt}\left(P.A\right)\right) \sqcap \Phi\left(R_{wrt}\left(Q.A\right)\right)\right) \qquad \text{[Definition 4.3.1]} \end{aligned}$$

thus internal choice in CSP is the linked counterpart of internal choice of actions in *Circus*.

Therefore, the linked internal choice of two processes in CSP has the same behaviour as the original internal choice in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 12 has the same semantics as internal choice in *Circus* and the rule is sound. □

5.7.4 Parallel Composition

Lemma 5.7.4 (Parallel Composition of Processes)**.** *Link Rule 13 for parallel composition of processes is sound.*

Proof. (1). The parallel composition of two processes in *Circus* is defined below

$$P \llbracket cs \rrbracket Q \mathrel{\widehat=} \left(\begin{array}{l} \textbf{begin}\,\textbf{state}\; State == P.StPar \land Q.StPar \\ \qquad P.Pars \land_{\Xi} Q.StPar \\ \qquad Q.Pars \land_{\Xi} P.StPar \\ \qquad \bullet\; P.A \llbracket \alpha(P.StPar) \mid cs \mid \alpha(Q.StPar) \rrbracket Q.A \\ \textbf{end} \end{array} \right)$$

[Oliveira [35, Definition B.43]]

where the main action of parallel composition of two processes in cs is a parallel composition of the main action of P and the main action of Q in cs.

(2). The parallel composition of two processes is linked to a $CSP \parallel_B Z$ program $P \underset{\Phi(cs)}{\parallel} Q$ by Link Rule 13.

Since the states, the schemas and the main actions of P and Q have been linked by Link Rule 9, this rule only links parallel composition. And

$$P \underset{\Phi(cs)}{\parallel} Q = \Phi\left(R_{wrt}\left(P.A\right)\right) \underset{\Phi(cs)}{\parallel} \Phi\left(R_{wrt}\left(Q.A\right)\right)$$

because

$$\begin{aligned} P &= \Phi\left(R_{wrt}\left(P.A\right)\right) \\ Q &= \Phi\left(R_{wrt}\left(Q.A\right)\right) && \text{[Link Rule 9]} \end{aligned}$$

(3). Additionally, according to Link Rule 43,

a) provided this parallel composition of processes, $(P \llbracket cs \rrbracket Q)$, is not a process in external choice, then

$$\begin{aligned} & \Phi\left(R_{wrt}\left(P.A \llbracket \alpha(P.StPar) \mid cs \mid \alpha(Q.StPar) \rrbracket Q.A\right)\right) \\ &= \Phi\left(R_{mrg}\left(R_{pre}\left(P.A\right), R_{pre}\left(Q.A\right)\right)\right) \rightarrow \left(\begin{array}{l} \Phi\left(R_{post}\left(P.A\right)\right) \\ \underset{\Phi(cs)}{\parallel} \\ \Phi\left(R_{post}\left(Q.A\right)\right) \end{array} \right) && \text{[Link Rule 43]} \\ &= \left(\begin{array}{l} \left(\Phi\left(R_{pre}\left(P.A\right)\right) \rightarrow \Phi\left(R_{post}\left(P.A\right)\right)\right) \\ \underset{\Phi(cs)}{\parallel} \\ \left(\Phi\left(R_{pre}\left(Q.A\right)\right) \rightarrow \Phi\left(R_{post}\left(Q.A\right)\right)\right) \end{array} \right) \\ & && \text{[Hoare [7, Section 2.2.1, L4], Definition 4.3.1 and Definition 4.3.2]} \\ &= \left(\begin{array}{l} \left(\Phi\left(R_{wrt}\left(P.A\right)\right)\right) \\ \underset{\Phi(cs)}{\parallel} \\ \left(\Phi\left(R_{wrt}\left(Q.A\right)\right)\right) \end{array} \right) && \text{[Definition 4.3.1]} \end{aligned}$$

b) provided this parallel composition of processes, $(P \llbracket cs \rrbracket Q)$, is a process in external choice, then P and Q have to be PA according to R_{wrt} Rule 22 and both $R_{pre}\left(P.A\right)$ and $R_{pre}\left(Q.A\right)$ are empty because of Definition B.4.1. Then

$$\Phi\left(R_{wrt}\left(P.A \llbracket \alpha(P.StPar) \mid cs \mid \alpha(Q.StPar) \rrbracket Q.A\right)\right)$$

$$= \Phi\left(R_{mrg}\left(R_{pre}\left(P.A\right), R_{pre}\left(Q.A\right)\right)\right) \to \left(\begin{array}{c}\Phi\left(R_{post}\left(P.A\right)\right) \\ \underset{\Phi(cs)}{\parallel} \\ \Phi\left(R_{post}\left(Q.A\right)\right)\end{array}\right) \qquad \text{[Link Rule 43]}$$

$$= \left(\begin{array}{c}\Phi\left(R_{post}\left(P.A\right)\right) \\ \underset{\Phi(cs)}{\parallel} \\ \Phi\left(R_{post}\left(Q.A\right)\right)\end{array}\right) \qquad \text{[Both } R_{pre} \text{ are empty]}$$

$$= \left(\begin{array}{c}\Phi\left(R_{wrt}\left(P.A\right)\right) \\ \underset{\Phi(cs)}{\parallel} \\ \Phi\left(R_{wrt}\left(Q.A\right)\right)\end{array}\right) \qquad \text{[Definition 4.3.1]}$$

thus parallel composition in CSP is the linked counterpart of parallel composition of actions in *Circus*.

Therefore, the linked parallel composition of two processes in CSP has the same behaviour as the original parallel composition in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 13 has the same semantics as parallel composition in *Circus* and the rule is sound. □

5.7.5 Interleaving

Lemma 5.7.5 (Interleaving of Processes)**.** *Link Rule 14 for interleaving of processes is sound.*

Proof. (1). The interleaving of two processes in *Circus* is defined below

$$P \mathbin{|||} Q \mathrel{\widehat=} \left(\begin{array}{l}\textbf{begin state}\ State == P.StPar \land Q.StPar \\ \qquad P.Pars \land_{\Xi} Q.StPar \\ \qquad Q.Pars \land_{\Xi} P.StPar \\ \qquad \bullet\ P.A \mathbin{[\![} \alpha(P.StPar) \mid \alpha(Q.StPar) \mathbin{]\!]} Q.A \\ \textbf{end}\end{array}\right)$$

[Oliveira [35, Definition B.44]]

where the main action of interleave of two processes is an interleave of the main action of P and the main action of Q.

(2). The interleave of two processes is linked to a $CSP \parallel_B Z$ program $P \mathbin{|||} Q$ by Link Rule 14.

Since the states, the schemas and the main actions of P and Q have been linked by Link Rule 9, this rule only links parallel composition. And

$$P \mathbin{|||} Q = \Phi\left(R_{wrt}\left(P.A\right)\right) \mathbin{|||} \Phi\left(R_{wrt}\left(Q.A\right)\right)$$

because

$$P = \Phi\left(R_{wrt}\left(P.A\right)\right)$$
$$Q = \Phi\left(R_{wrt}\left(Q.A\right)\right) \qquad \text{[Link Rule 9]}$$

(3). Additionally, according to Link Rule 45,

a) provided this interleave of processes, $(P \mathbin{|||} Q)$, is not a process in external choice, then

$$\Phi\left(R_{wrt}\left(P.A \mathbin{[\![} \alpha(P.StPar) \mid \alpha(Q.StPar) \mathbin{]\!]} Q.A\right)\right)$$

$$= \Phi\left(R_{mrg}\left(R_{pre}\left(P.A\right), R_{pre}\left(Q.A\right)\right)\right) \to \left(\begin{array}{c}\Phi\left(R_{post}\left(P.A\right)\right) \\ ||| \\ \Phi\left(R_{post}\left(Q.A\right)\right)\end{array}\right) \qquad [\Phi \text{ Rule 45}]$$

$$= \begin{pmatrix} (\Phi\,(R_{pre}\,(P.A)) \to \Phi\,(R_{post}\,(P.A))) \\ ||| \\ (\Phi\,(R_{pre}\,(Q.A)) \to \Phi\,(R_{post}\,(Q.A))) \end{pmatrix}$$

[Hoare [7, Section 3.6.1, L7], Definition 4.3.1 and Definition 4.3.2]

$$= \begin{pmatrix} (\Phi\,(R_{wrt}\,(P.A))) \\ ||| \\ (\Phi\,(R_{wrt}\,(Q.A))) \end{pmatrix} \qquad \text{[Definition 4.3.1]}$$

b) provided this interleave of processes, $(P \mathbin{|||} Q)$, is a process in external choice, then P and Q have to be PA according to R_{wrt} Rule 22 and both $R_{pre}\,(P.A)$ and $R_{pre}\,(Q.A)$ are empty because of Definition B.4.1. Then

$$\Phi\,(R_{wrt}\,(P.A \mathbin{[\![} \alpha(P.StPar) \mid \alpha(Q.StPar) \mathbin{]\!]} Q.A))$$

$$= \Phi\,(R_{mrg}\,(R_{pre}\,(P.A)\,, R_{pre}\,(Q.A))) \to \begin{pmatrix} \Phi\,(R_{post}\,(P.A)) \\ ||| \\ \Phi\,(R_{post}\,(Q.A)) \end{pmatrix} \qquad [\Phi \text{ Rule 45}]$$

$$= \begin{pmatrix} \Phi\,(R_{post}\,(P.A)) \\ ||| \\ \Phi\,(R_{post}\,(Q.A)) \end{pmatrix} \qquad [\text{Both } R_{pre} \text{ are empty}]$$

$$= \begin{pmatrix} \Phi\,(R_{wrt}\,(P.A)) \\ ||| \\ \Phi\,(R_{wrt}\,(Q.A)) \end{pmatrix} \qquad \text{[Definition 4.3.1]}$$

thus interleave in CSP is the linked counterpart of interleave of actions in *Circus*.

Therefore, the linked interleave of two processes in CSP has the same behaviour as the original interleave in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 14 has the same semantics as interleave in *Circus* and the rule is sound. □

5.7.6 Hiding

Lemma 5.7.6 (Hiding of Processes)**.** *Link Rule 15 for hiding of a process is sound.*

Proof. (1). The event hiding from a process in *Circus* is defined below

$$(P) \setminus cs = \begin{pmatrix} \textbf{begin state}\; State == P.StPar \\ \quad P.Pars == [\cdots] \\ \quad \bullet\; P.A \setminus cs \\ \textbf{end} \end{pmatrix} \qquad \text{[Oliveira [35, Definition B.45]]}$$

where the main action of hiding events cs from a process is an action like its main action but from which all events in cs are hidden.

(2). The event hiding from a processes is linked to a $CSP \parallel_B Z$ program $P \setminus \Phi(cs)$ by Link Rule 15.

Since the states, the schemas and the main action of P have been linked by Link Rule 9, this rule only links event hiding. And

$$P \setminus \Phi(cs) = \Phi\,(R_{wrt}\,(P.A)) \setminus \Phi\,(cs)$$

because

$$P = \Phi\,(R_{wrt}\,(P.A)) \qquad \text{[Link Rule 9]}$$

(3). Additionally, according to Link Rule 47,

a) provided this event hiding operator of a process, $P \setminus cs$, is not a process in external choice, then

$$\begin{aligned}
&\Phi\left(R_{wrt}\left(P.A \setminus cs\right)\right)\\
&= \Phi\left(R_{Pre}\left(P.A\right)\right) \rightarrow \left(\Phi\left(R_{Post}\left(P.A\right)\right) \setminus \Phi\left(cs\right)\right) && \text{[Link Rule 47]}\\
&= \left(\Phi\left(R_{Pre}\left(P.A\right)\right) \rightarrow \Phi\left(R_{Post}\left(P.A\right)\right)\right) \setminus \Phi\left(cs\right)\\
& && \text{[Hoare [7, Section 3.5.1, L5] and Definition 4.3.1]}\\
&= \Phi\left(R_{wrt}\left(P.A\right)\right) \setminus \Phi\left(cs\right) && \text{[Definition 4.3.1]}
\end{aligned}$$

b) provided this event hiding operator of a process, $P \setminus cs$, is a process in external choice, then P has to be PA according to R_{wrt} Rule 22 and $R_{pre}\left(P.A\right)$ is empty because of Definition B.4.1. Then

$$\begin{aligned}
&\Phi\left(R_{wrt}\left(P.A \setminus cs\right)\right)\\
&= \Phi\left(R_{Pre}\left(P.A\right)\right) \rightarrow \left(\Phi\left(R_{Post}\left(P.A\right)\right) \setminus \Phi\left(cs\right)\right) && \text{[Link Rule 47]}\\
&= \left(\Phi\left(R_{Post}\left(P.A\right)\right) \setminus \Phi\left(cs\right)\right) && [R_{pre} \text{ is empty}]\\
&= \left(\Phi\left(R_{wrt}\left(P.A\right)\right) \setminus \Phi\left(cs\right)\right) && \text{[Definition 4.3.1]}
\end{aligned}$$

thus the event hiding in CSP is the linked counterpart of the event hiding from an action in *Circus*.

Therefore, the linked event hiding from a process in CSP has the same behaviour as the original event hiding from a process in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 15 has the same semantics as the event hiding from a process in *Circus* and the rule is sound. □

5.7.7 Unnamed Parametrised Process Invocation

Lemma 5.7.7 (Unnamed Parametrised Process Invocation)**.** *Link Rule 16 for unnamed parametrised process invocation is sound.*

Proof. An unnamed parametrised process invocation is rewritten to a named parametrised process and its invocation by Link Rule 16. Then they are linked by the rules of the parametrised process and its invocation. Its soundness is the same as that of the parametrised process and its invocation (Section 5.8). □

5.7.8 Parametrised Process Invocation

Lemma 5.7.8 (Parametrised Process Invocation)**.** *Link Rule 17 for parametrised process invocation is sound.*

Proof. Refer to Section 5.8. □

5.7.9 Process Invocation

Lemma 5.7.9 (Process Invocation)**.** *Link Rule 18 for process invocation is sound.*

Proof. (1). The semantics of a process invocation, that is, a reference to a process name, is given by the copy rule: it is the body of the process.

$$P = \left(\begin{array}{l}
\mathbf{begin}\\
\quad \mathbf{state}\ StPar == [\, s_1 : T_1 \,; \cdots s_n : T_n \mid p \,]\\
\quad Init == [\, (StPar)' \mid pi \,]\\
\quad Pars == [\cdots]\\
\quad \bullet\ A\\
\mathbf{end}
\end{array}\right)$$

where the main action of a process invocation is the main action of this process.

(2). A process invocation in *Circus* is linked to a $CSP \parallel_B Z$ program P by Link Rule 18. Since the states, the schemas and the main action of P have been linked by Link Rule 9, this rule only links the reference to the process. And

$$P = \Phi\left(R_{wrt}\left(P.A\right)\right)$$

because

$$P = \Phi\left(R_{wrt}\left(P.A\right)\right) \qquad \text{[Link Rule 9]}$$

(3). Therefore, the linked process P in CSP has the same behaviour as the original process invocation in *Circus* as well as the same state part which is stated in Section 5.6. In sum, the $CSP \parallel_B Z$ program got according to Link Rule 18 has the same semantics as the process invocation in *Circus* and the rule is sound. □

5.7.10 Unnamed Indexed Process Invocation

Lemma 5.7.10 (Unnamed Indexed Process Invocation)**.** *Link Rule 19 for unnamed indexed process invocation is sound.*

Proof. An unnamed indexed process invocation is rewritten to a named indexed process and its invocation by Link Rule 19. Then they are linked by the rules of the indexed process and its invocation. Its soundness is the same as that of the indexed process and its invocation (Section 5.9). □

5.7.11 Indexed Process Invocation

Lemma 5.7.11 (Indexed Process Invocation)**.** *Link Rule 20 for indexed process invocation is sound.*

Proof. Refer to Section 5.9. □

5.7.12 Renaming Operator

Lemma 5.7.12 (Renaming Operator)**.** *Link Rule 21 for process renaming operator is sound.*

Proof. The renaming operator $P[c_{old} := c_{new}]$ is simply the syntactic renaming of the old channels to the new channels in a process. □

5.7.13 Iterated Sequential Composition

Lemma 5.7.13 (Iterated Sequential Composition)**.** *Link Rule 24 for iterated sequential composition of processes is sound.*

Proof. (1). The semantics of the iterated operators in *Circus* is given by the expansion of the operators. Therefore,

$$; x : T \bullet P(x) \mathrel{\widehat{=}} P(x_1) ; \cdots ; P(x_n)$$

where T is a set with n elements: $x_1, \ldots, x_n$.

(2). The iterated sequential composition is linked to a $CSP \parallel_B Z$ program

$$;_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right)$$

by Link Rule 24.

The replicated operator in CSP denotes the expansion of the operator as well. Thus,

$$;_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) \mathrel{\widehat=} \Phi\left(R_{wrt}\left(P(x_1)\right)\right) ; \cdots ; \Phi\left(R_{wrt}\left(P(x_n)\right)\right)$$

(3). Since $(\Phi\left(R_{wrt}\left(P\right)\right) ; \Phi\left(R_{wrt}\left(Q\right)\right))$ is a sound link of $(P\,;\,Q)$ in *Circus* to CSP (see Section 5.7.1), the expanded sequential composition of processes in CSP in (2) has the same semantics as the corresponding expanded sequential composition of processes in *Circus* in (1). Finally, the replicated sequential composition in CSP is a sound link of the iterated sequential composition in *Circus*, and the rule is sound. □

5.7.14 Iterated External Choice

Lemma 5.7.14 (Iterated External Choice)**.** *Link Rule 25 for iterated external choice of processes is sound.*

Proof. (1). The semantics of the iterated operators in *Circus* is given by the expansion of the operators. Therefore,

$$\Box\, x : T \bullet PA(x) \mathrel{\widehat=} PA(x_1) \Box \cdots \Box PA(x_n)$$

where PA is a prefixed process.

(2). The iterated external choice is linked to a $CSP \parallel_B Z$ program

$$\Box_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(PA(x)\right)\right)$$

by Link Rule 25.

The replicated operator in CSP denotes the expansion of the operator as well. Thus,

$$\Box_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(PA(x)\right)\right) \mathrel{\widehat=} \Phi\left(R_{wrt}\left(PA(x_1)\right)\right) \Box \cdots \Box \Phi\left(R_{wrt}\left(PA(x_n)\right)\right)$$

(3). Since $(\Phi\left(R_{wrt}\left(PA_1\right)\right) \Box \Phi\left(R_{wrt}\left(PA_2\right)\right))$ is a sound link of $(PA_1 \Box PA_2)$ in *Circus* to CSP (see Section 5.7.2), the expanded external choice of processes in CSP in (2) has the same semantics as the corresponding expanded external choice of processes in *Circus* in (1). Finally, the replicated external choice in CSP is a sound link of the iterated external choice in *Circus* if the process is a PA, and the rule is sound. □

5.7.15 Iterated Internal Choice

Lemma 5.7.15 (Iterated Internal Choice)**.** *Link Rule 26 for iterated internal choice of processes is sound.*

Proof. (1). The semantics of the iterated operators in *Circus* is given by the expansion of the operators. Therefore,

$$\sqcap\, x : T \bullet P(x) \mathrel{\widehat=} P(x_1) \sqcap \cdots \sqcap P(x_n)$$

(2). The iterated internal choice is linked to a $CSP \parallel_B Z$ program

$$\sqcap_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right)$$

by Link Rule 26.

The replicated operator in CSP denotes the expansion of the operator as well. Thus,

$$\sqcap_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) \mathrel{\widehat=} \Phi\left(R_{wrt}\left(P(x_1)\right)\right) \sqcap \cdots \sqcap \Phi\left(R_{wrt}\left(P(x_n)\right)\right)$$

(3). Since $(\Phi\left(R_{wrt}\left(P\right)\right) \sqcap \Phi\left(R_{wrt}\left(Q\right)\right))$ is a sound link of $(P \Box Q)$ in *Circus* to CSP (see Section 5.7.3), the expanded internal choice of processes in CSP in (2) has the same semantics as the corresponding expanded internal choice of processes in *Circus* in (1). Finally, the replicated internal choice in CSP is a sound link of the iterated internal choice in *Circus*, and the rule is sound. □

5.7.16 Iterated Parallel Composition

Lemma 5.7.16 (Iterated Parallel Composition)**.** *Link Rule 27 for iterated parallel composition of processes is sound.*

Proof. (1). The semantics of the iterated operators in *Circus* is given by the expansion of the operators. Therefore,

$$[\![\ CS \ ]\!] \ x : T \bullet P(x) \mathrel{\widehat=} P(x_1) \ [\![\ CS \ ]\!] \ (P(x_2) \ [\![\ CS \ ]\!] \cdots [\![\ CS \ ]\!] \ P(x_n))$$

(2). The iterated parallel composition is linked to a $CSP \parallel_B Z$ program

$$\underset{\Phi(CS)}{\big\|} \ {}_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right)$$

by Link Rule 27.

The replicated operator in CSP denotes the expansion of the operator as well. Thus,

$$\begin{aligned}
&\underset{\Phi(CS)}{\big\|} \ {}_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) \\
&\mathrel{\widehat=} \Phi\left(R_{wrt}\left(P(x_1)\right)\right) \underset{\Phi(CS)}{\|} \left(\Phi\left(R_{wrt}\left(P(x_2)\right)\right) \underset{\Phi(CS)}{\|} \cdots \underset{\Phi(CS)}{\|} \Phi\left(R_{wrt}\left(P(x_n)\right)\right)\right)
\end{aligned}$$

(3). Since $\left(\Phi\left(R_{wrt}\left(P\right)\right) \underset{\Phi(CS)}{\|} \Phi\left(R_{wrt}\left(Q\right)\right)\right)$ is a sound link of $(P \ [\![\ CS \ ]\!] \ Q)$ in *Circus* to CSP (see Section 5.7.4), the expanded parallel composition of processes in CSP in (2) has the same semantics as the corresponding expanded parallel composition of processes in *Circus* in (1). Finally, the replicated parallel composition in CSP is a sound link of the iterated parallel composition in *Circus*, and the rule is sound. □

5.7.17 Iterated Interleaving

Lemma 5.7.17 (Iterated Interleaving)**.** *Link Rule 28 for iterated interleaving of processes is sound.*

Proof. (1). The semantics of the iterated operators in *Circus* is given by the expansion of the operators. Therefore,

$$|||\ x : T \bullet P(x) \mathrel{\widehat=} P(x_1) \ ||| \ (P(x_2) \ ||| \cdots ||| \ P(x_n))$$

(2). The iterated interleaving is linked to a $CSP \parallel_B Z$ program

$$|||_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right)$$

by Link Rule 28.

The replicated operator in CSP denotes the expansion of the operator as well. Thus,

$$\begin{aligned}
&|||_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) \\
&\mathrel{\widehat=} \Phi\left(R_{wrt}\left(P(x_1)\right)\right) \ ||| \ \left(\Phi\left(R_{wrt}\left(P(x_2)\right)\right) \ ||| \cdots ||| \ \Phi\left(R_{wrt}\left(P(x_n)\right)\right)\right)
\end{aligned}$$

(3). Since $\left(\Phi\left(R_{wrt}\left(P\right)\right) \ ||| \ \Phi\left(R_{wrt}\left(Q\right)\right)\right)$ is a sound link of $(P \ ||| \ Q)$ in *Circus* to CSP (see Section 5.7.5), the expanded interleaving of processes in CSP in (2) has the same semantics as the corresponding expanded interleaving of processes in *Circus* in (1). Finally, the replicated interleaving in CSP is a sound link of the iterated interleaving in *Circus*, and the rule is sound. □

5.8 Parametrised Process and its Invocation

Lemma 5.8.1 (Parametrised Process and its Invocation). *Link Rule 29 and 17 for parametrised process and its invocation are sound.*

Proof. (1). The semantics of a parametrised process in *Circus* is given by the semantics of parametrised procedure in UTP.

$$\begin{aligned}&(\textbf{process}\ PP \mathrel{\widehat=} x : T \bullet P)\\ &\mathrel{\widehat=} (PP = \{\!\!\{ \lambda\, x : var(T) \bullet P \}\!\!\}) && \text{[Hoare \& He [24, Section 9.2]]}\\ &= (PP = \{\!\!\{ \lambda\, x : val(T) \bullet P \}\!\!\}) && \text{[The parameter } x \text{ is not allowed to be changed]}\end{aligned}$$

Therefore, the parametrised process invocation $PP(ax)$ becomes

$$\begin{aligned}&PP(ax)\\ &= PP(x)[ax, ax'/x, x'] && \text{[Hoare \& He [24, Theorem 9.2.7]]}\\ &= \{\!\!\{P\}\!\!\}[ax, ax'/x, x']\\ &= P[ax/x] && \text{[For a value parameter having } val,\ x' = x]\end{aligned}$$

If ax is x_i, it becomes a process invocation $P[x_i/x]$.

(2). A parametrised process PP and its invocation are linked to a $CSP \parallel_B Z$ program by Link Rule 29 and Link Rule 17 respectively. For an invocation $PP(x_i)$, it is linked to $\Phi\,(P[x_i/x])$ by Link Rule 17.

(3). By comparing the parametrised process invocation in (1) and (2), we conclude the rules for the parametrised process and its invocation are sound. □

5.9 Indexed Process and its Invocation

Lemma 5.9.1 (Indexed Process and its Invocation). *Link Rule 30 and 20 for indexed process and its invocation are sound.*

Proof. (1). The semantics of an indexed process in *Circus* is given by the semantics of the parametrised process.

(2). An indexed process IP and its invocation are linked to a $CSP \parallel_B Z$ program by Link Rule 30 and Link Rule 20.

(3). The semantics of the indexed process and its invocation is the same as the parametrised process and its invocation except that all channels are renamed in the indexed process. Since the rules defined for the parametrised (Section 5.8) processes are sound, those rules are sound as well. □

5.10 Actions

5.10.1 Schema Expression as Action

Lemma 5.10.1 (Schema Expression as Action). *Link Rule 31 for schema expression as action is sound.*

Proof. (1). Schema expressions actually are transformed into specification statements by the basic conversion rule bC [89]. The precondition of this rule requires the schema shall specify an operation. Provided an operational schema Op

$$Op == [\, decl\, ;\, ins? : T_i\, ;\, outs! : T_o \mid pred_0\,]$$

has a $decl$, $ins?$ and $outs!$ in its declaration part and $pred_0$ in its predicate part. In order to convert it into a specification statement, it should be normalised in advance using the normalisation technique [63, p. 159]. In addition, input and output variables are simply the syntactic sugar for undashed and dashed variables respectively. Therefore, the schema Op is normalised and transformed to

$$Op_{norm} == [\, udecl \; ; ddecl' \mid pred \,]$$

where

- $udecl$ stands for undashed variables, including undashed state variables and input variables;
- $ddecl'$ stands for dashed variables, including dashed state variables and output variables.

Then this normalised schema is easily converted into a specification statement and, finally, the semantics of the schema expressions is embedded into the specification statements.

$$\begin{aligned}
&\left[udecl \; ; ddecl' \mid pred\right] \\
&\mathrel{\widehat=} ddecl : [\exists\, ddecl' \bullet pred, pred] && \text{[Oliveira [35, Definition B.40]]} \\
&= \mathbf{R}(\exists\, ddecl' \bullet pred \vdash pred \land \neg wait' \land tr' = tr) && \text{[Oliveira [35, Definition B.32]]} \\
&= \mathbf{R}(pre \vdash post \land \neg wait' \land tr' = tr)
\end{aligned}$$

where

- $ddecl$ stands for a comma-separated list of undashed variables which are introduced as dashed variables in $ddecl'$;
- pre is an abbreviation for $(\exists\, ddecl' \bullet pred)$;
- $post$ is for $pred$.

In sum, the schema expression action is successfully terminated with the postcondition is established only if the precondition holds. On termination, only the variables in the frame — introduced in $ddecl'$ — are changed and others are not changed. Furthermore, the trace is left unchanged. If the precondition does not hold, it is diverged like the reactive abortion **Chaos** .

(2). A schema expression in *Circus* is linked to a $CSP \parallel_B Z$ program

$$\left\{\begin{array}{l}
\Omega_3\left(\begin{array}{l}
P\_SExp \mathrel{\widehat=} [\, decl \; ; \Xi Q_1\_StPar \; ; \cdots \Xi Q_n\_StPar; \\
\qquad\qquad ins? : T_i \; ; outs! : T_o \mid p \,] \\
P\_SExp\_fOp \mathrel{\widehat=} [\Xi P\_StPar \; ; \Xi Q_1\_StPar \; ; \cdots \Xi Q_n\_StPar; \\
\qquad\qquad ins? : T_i \mid \neg\, \mathbf{pre}\, P\_SExp]
\end{array}\right) \\
\\
\left(\begin{array}{l}
channel\ P\_SExp : \Phi(T_i).\Phi(T_o) \\
channel\ P\_SExp\_fOp : \Phi(T_i) \\
HIDE\_CSPB = \{\!| P\_SExp, P\_SExp\_fOp |\!\} \\
P\_SExp!ins?outs \rightarrow SKIP \;\Box\; P\_SExp\_fOp!ins \rightarrow \mathbf{div}
\end{array}\right)
\end{array}\right.$$

by Link Rule 31.

According to Theorem 5.2.1, its semantics is

$$\mathbf{R}\left(pre \vdash post \land \neg wait' \land tr' = tr\right)$$

(3). Since the P_SExp schema in Link Rule 31 is a linked counterpart of the corresponding schema in *Circus* and only transformed by the renaming rule (R_{wrt} Rule 24) and the state merge rule (Ω_1 Rule 1). Therefore, the precondition and postcondition of this transformed schema are the same as their counterparts in *Circus*. Finally, compared to the semantics of schema expressions in (1), the linked $CSP \parallel_B Z$ programs have the same semantics in (2), and the rule is sound. □

5.10.2 CSP Actions

5.10.2.1 Basic Actions

Lemma 5.10.2 (Basic Actions)**.** *Link Rule 32 for basic actions is sound.*

Proof. According to Lemma 5.10.3, Lemma 5.10.4, and Lemma 5.10.5, Link Rule 32 is sound for **Skip**, **Stop**, and **Chaos**. That is, Link Rule 32 for basic actions is sound. □

Lemma 5.10.3 (**Skip**)**.** *Link Rule 32 for* **Skip** *is sound.*

Proof. (1). The semantics of the **Skip** action in *Circus* is defined as

$$\mathbf{Skip} \mathrel{\widehat=} \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \land \neg wait' \land v' = v) \qquad \text{[Oliveira [35, Definition B.3]]}$$

(2). The **Skip** action in *Circus* is linked to the $SKIP$ process in CSP by Link Rule 32. The semantics of the $SKIP$ process in CSP is defined as

$$SKIP \mathrel{\widehat=} \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \land \neg wait') \qquad \text{[Cavalcanti \& Woodcock [68, Law 134]]}$$

When considering this $SKIP$ process is in the $CSP \parallel B$ model (Equation 2.2), according to Law 2.4.1, the semantics of the $SKIP$ process in $CSP \parallel B$ model is

$$SKIP \mathrel{\widehat=} \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \land \neg wait' \land v' = v)$$
$$\text{[Cavalcanti \& Woodcock [68, Law 134] and Law 2.4.1]}$$

(3). Therefore, both **Skip** in *Circus* has the same semantics as $SKIP$ in $CSP \parallel B$. Finally, the link is sound. □

Lemma 5.10.4 (**Stop**)**.** *Link Rule 32 for* **Stop** *is sound.*

Proof. (1). The semantics of the **Stop** action in *Circus* is defined as

$$\mathbf{Stop} \mathrel{\widehat=} \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \land wait') \qquad \text{[Oliveira [35, Definition B.2]]}$$

It is worth noting that **Stop** leaves the state variables unconstrained or loose in order to be in the unit for the external choice of actions due to the semantics of external choice that state changes would not resolve it. That is explained in Section 3.5 of [35].

(2). The **Stop** action in *Circus* is linked to the $STOP$ process in CSP by Link Rule 32. The semantics of the $STOP$ process in CSP is defined as

$$STOP = \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \land wait') \qquad \text{[Cavalcanti \& Woodcock [68, Section 6.3]]}$$

When considering this $STOP$ process is in the $CSP \parallel B$ model (Equation 2.2), according to Law 2.4.1, the semantics of the $STOP$ process in $CSP \parallel B$ model is

$$STOP \mathrel{\widehat=} \boldsymbol{R}(\mathbf{true} \vdash tr' = tr \land wait' \land v' = v)$$
$$\text{[Cavalcanti \& Woodcock [68, Section 6.3] and Law 2.4.1]}$$

(3). From the behavioural aspect, **Stop** and $STOP$ have the same semantics. But from the state aspect, $STOP$ is different from **Stop** which leaves state unconstrained. However in the real situation, deadlock is always not what the system engineer expects. When it deadlocks, a counterexample is generated and model checking fails. □

Lemma 5.10.5 (**Chaos**)**.** *Link Rule 32 for* **Chaos** *is sound.*

Proof. (1). The semantics of the **Chaos** action is shown as below.

$$\mathbf{Chaos} = \boldsymbol{R}(\mathbf{false} \vdash \mathbf{true}) \qquad \text{[Oliveira [35, Definition B.2]]}$$

(2). The **Chaos** action in *Circus* is linked to the **div** process in CSP by Link Rule 32. The **div** [71] in CSP is the worst process that does nothing but diverges. It is equal to $CHAOS$ in [68] and [24]. The semantics is

$$\mathbf{div} = \boldsymbol{R}(\mathbf{false} \vdash \mathbf{true}) \qquad \text{[Cavalcanti \& Woodcock [68]]}$$

(3). Therefore, **Chaos** and **div** have the same semantics and the link is sound. □

5.10.2.2 Prefixing

Synchronisation Channel

Lemma 5.10.6 (Synchronisation Channel)**.** *Link Rule 33 for prefixing with synchronisation channel, only having a channel name and no message communicated, is sound.*

For a prefixing process with a channel name only and without message communicated: $c \rightarrow \textbf{Skip}$,

Proof. (1). A prefixing $c \rightarrow \textbf{Skip}$ in *Circus* is defined as a reactive design

$$c \rightarrow \textbf{Skip} \mathrel{\widehat=} \boldsymbol{R}(\textbf{true} \vdash do_{\mathcal{C}}(c, Sync) \wedge v' = v) \qquad \text{[Oliveira [35, Definition B.10]]}$$

where

$$do_{\mathcal{C}}(c, e) \mathrel{\widehat=} \left(\left(tr' = tr\right) \wedge (c, e) \notin ref'\right) \lhd wait' \rhd \left(tr' = tr^\frown\langle (c, e)\rangle\right)$$
$$\text{[Oliveira [35, Definition B.9]]}$$

c is the channel name and e is the value communicated in channel c. *Sync* is a special case of communication value for synchronization and the communication pair $(c, Sync)$ represents the communication c without value only.

(2). The prefixing $c \rightarrow \textbf{Skip}$ in *Circus* is linked to the $CSP \parallel_B Z$ program

$$c \rightarrow \Phi\left(R_{wrt}\left(\textbf{Skip}\right)\right) = c \rightarrow SKIP \qquad \text{[Link Rule 32]}$$

by Link Rule 33.

The prefixing $c \rightarrow SKIP$ in CSP is defined by UTP semantics.

$$\begin{aligned}
& c \rightarrow SKIP \\
& \mathrel{\widehat=} \textbf{CSP1}\left(ok' \wedge do_{\mathcal{A}}(c)\right) && \text{[Hoare \& He [24, p. 209]]} \\
& = \textbf{CSP1}\left(ok' \wedge \boldsymbol{R}\left(\textbf{true} \vdash \left(tr' = tr \wedge c \notin ref'\right) \lhd wait' \rhd \left(tr' = tr^\frown\langle c\rangle\right)\right)\right) \\
& && \text{[Equation 5.1]} \\
& = \boldsymbol{R}\left(\textbf{true} \vdash \left(tr' = tr \wedge c \notin ref'\right) \lhd wait' \rhd \left(tr' = tr^\frown\langle c\rangle\right)\right) \\
& && \text{[Reactive-Design-CSP1 [68, Law 121]]} \\
& = \boldsymbol{R}\left(\textbf{true} \vdash \left(\left(tr' = tr \wedge c \notin ref'\right) \lhd wait' \rhd \left(\left(tr' = tr^\frown\langle c\rangle\right)\right) \wedge v' = v\right)\right) \\
& && \text{[State variables in B are not changed and Law 2.4.2]}
\end{aligned}$$

where

$$\begin{aligned}
do_{\mathcal{A}}(c) & \mathrel{\widehat=} \Phi(c \notin ref' \lhd wait' \rhd tr' = tr^\frown\langle c\rangle) && \text{[Hoare \& He [24, p. 200]]} \\
& = \boldsymbol{R}(((tr' = tr) \wedge wait' \vee (tr < tr')) \wedge (c \notin ref' \lhd wait' \rhd tr' = tr^\frown\langle c\rangle)) \\
& && \text{[Hoare \& He [24, p. 199]]} \\
& = \boldsymbol{R}(tr' = tr \wedge c \notin ref' \lhd wait' \rhd tr' = tr^\frown\langle c\rangle) && (5.1) \\
& = \boldsymbol{R}(\textbf{true} \vdash tr' = tr \wedge c \notin ref' \lhd wait' \rhd tr' = tr^\frown\langle c\rangle) && \text{[Reactive Design]}
\end{aligned}$$

Note that Φ [24, p. 199] in the equation is different from our Φ link function.

(3). Therefore, the semantics of the prefixing $c \rightarrow \textbf{Skip}$ is the same as that of the prefixing $c \rightarrow SKIP$ in $CSP \parallel_B Z$. The rule is sound for $c \rightarrow \textbf{Skip}$. □

For a prefixing process with a channel name only and without message communicated: $c \rightarrow A$, *Proof.* (1). The prefixing $c \rightarrow A$ in *Circus* is defined as

$$c \rightarrow A \mathrel{\widehat{=}} (c \rightarrow \textbf{Skip})\ ;\ A \qquad \text{[Oliveira [35, Definition B.13]]}$$

(2). The prefixing $c \rightarrow A$ in *Circus* is linked to the $CSP \parallel_B Z$ program

$$c \rightarrow \Phi\left(R_{wrt}\left(A\right)\right)$$

by Link Rule 33. It is equal to

$$\left(c \rightarrow SKIP\right)\ ;\ \Phi\left(R_{wrt}\left(A\right)\right) \qquad \text{[Hoare \& He [24, Definition 8.2.5]]}$$

(3). Since ; and $\Phi\left(R_{wrt}\left(A\right)\right)$ are the linked counterparts of ; and A in *Circus* and $c \rightarrow SKIP$ has the same semantics as $c \rightarrow \textbf{Skip}$ as proven above, finally, the rule for the prefixing $c \rightarrow A$ is sound. ☐

Output Channel

Lemma 5.10.7 (Output Channel without State Components evaluated)**.** *Link Rule 34 for prefixing with output channel is sound.*

For the prefixing $c.e \rightarrow \textbf{Skip}$ or $c!e \rightarrow \textbf{Skip}$ in *Circus*, provided e does not evaluate state variables,

Proof. The prefixing is linked to $c.e \rightarrow SKIP$ or $c!e \rightarrow SKIP$ by Link Rule 34. $c.e$ or $c!e$ can be treated as the synchronization channel $c.e$ without value or the channel c with the value e. In both cases, they have the same semantics in *Circus* and CSP. ☐

Lemma 5.10.8 (Output Channel with State Components evaluated)**.** *Link Rule 35 for prefixing with output channel is sound.*

For the prefixing $c.e \rightarrow \textbf{Skip}$ or $c!e \rightarrow \textbf{Skip}$ in *Circus*, provided e does evaluate state variables $s_i, \cdots, s_j$,

Proof. (1). For the prefixing $c.e(s_i, \cdots, s_j) \rightarrow \textbf{Skip}$ in *Circus*, its semantics defined below outputs the value of the expression $e(s_i, \cdots, s_j)$ on the channel c and after that it terminates.

$$c.e(s_i, \cdots, s_j) \rightarrow \textbf{Skip} \mathrel{\widehat{=}} \boldsymbol{R}(\textbf{true} \vdash do_{C}(c, e(s_i, \cdots, s_j)) \wedge v' = v)$$
[Oliveira [35, Definition B.10]]

(2). For the prefixing $c.e(s_i, \cdots, s_j) \rightarrow \textbf{Skip}$, it is linked to the $CSP \parallel_B Z$ program

$$\begin{cases} channel\ P\_Op\_s_i : \Phi(T_{s_i}) \\ \cdots \\ channel\ P\_Op\_s_j : \Phi(T_{s_j}) \\ HIDE\_CSPB = \{\!|P\_Op\_s_i, \cdots, P\_Op\_s_j, \cdots|\!\} \\ P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow c.\Phi\left(e(s_i, \cdots, s_j)\right) \rightarrow SKIP \end{cases}$$

by Link Rule 35.

As a $CSP \parallel B$ program, it is given by Definition 2.4.1.

$$\left(Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow c.\Phi\left(e(s_i, \cdots, s_j)\right) \rightarrow SKIP\right) \underset{ops}{{}_{C}\parallel_{B}} (B)_{v}^{v'}$$

$$= \left\{ \begin{array}{l} \exists\, v_0 \bullet \\ \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \\ \quad \left(\begin{array}{l} (\cdots \rightarrow Op\_s_j?s_j \rightarrow c.\Phi\,(e(s_i,\cdots,s_j)) \rightarrow SKIP)\,[\![s_i/s_i]\!] \\ {}_{C}\|_{B}\ (B)^{v'}_{v_0} \\ \quad ops \end{array} \right) \end{array} \right) \\ \wedge_R\ pred(Op\_s_i)[v_0/v'] \end{array} \right.$$

[Law 2.4.3]

$$= \left(Op\_s_i?s_i \rightarrow \left(\begin{array}{l} (\cdots \rightarrow Op\_s_j?s_j \rightarrow c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i/s_i]\!] \rightarrow SKIP) \\ {}_{C}\|_{B}\ (B)^{v'}_{v} \\ \quad ops \end{array} \right) \right)$$

[$v_0 = v$ because $v' = v$ for Op_s_i in R_{wrt} Rule 23]

$$= \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \quad \left((c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i,\cdots,s_j/s_i,\cdots,s_j]\!] \rightarrow SKIP)\ \underset{ops}{{}_{C}\|_{B}}\ (B)^{v'}_{v} \right) \end{array} \right)$$

[Further application of the same law]

$$= \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \quad c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i,\cdots,s_j/s_i,\cdots,s_j]\!] \rightarrow \left(SKIP\ \underset{ops}{{}_{C}\|_{B}}\ (B)^{v'}_{v} \right) \end{array} \right)$$

[Law 2.4.2]

$$= \left\{ \begin{array}{l} \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \quad c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i,\cdots,s_j/s_i,\cdots,s_j]\!] \rightarrow SKIP \end{array} \right) \\ \wedge_R\ (v' = v) \end{array} \right. \qquad \text{[Law 2.4.1]}$$

where $[\![\,]\!]$, such as $e[\![s_i/s_i]\!]$ or $P[\![s_i/s_i]\!]$, is a special notation introduced to denote the substitution of the variable s_i in the expression e or the variable s_i to be evaluated in the first construct of P by its value (the same name s_i). For example, the current value of the variable s_1 is 1, then

$$(s_1 + s_1 * 2)\,[\![s_1/s_1]\!] = 1 + 1 * 2$$
$$(c.s_1 \rightarrow SKIP)\,[\![s_1/s_1]\!] = c.1 \rightarrow SKIP$$
$$(c?s_1.s_1 \rightarrow SKIP)\,[\![s_1/s_1]\!] = c?s_1.1 \rightarrow SKIP$$

[The first input variable s_1 is not to be evaluated]

In addition, this notation is only used in CSP and not in *Circus*. Therefore, $A[\![s_i/s_i]\!]$ is not correct and $\Phi\,(R_{wrt}\,(A))\,[\![s_i/s_i]\!]$ is correct. And the notation is introduced just for reasoning only and will not be used in other places.

When taking $HIDE\_CSPB = \{\!|Op\_s_i,\cdots,Op\_s_j,\ldots|\!\}$ into account, it is simplified further.

$$\left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \quad c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i,\cdots,s_j/s_i,\cdots,s_j]\!] \rightarrow SKIP \end{array} \right) \setminus \{\!|Op\_s_i,\cdots,Op\_s_j|\!\}$$
$$= (c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i,\cdots,s_j/s_i,\cdots,s_j]\!] \rightarrow SKIP)$$

[Hiding [7, Section 3.5.1, L5]]

Finally, in the linked expression $\Phi\,(e)$, each occurrence of s_i,..., or s_j is substituted by its current value, then this linked expression is outputted from the channel c. Furthermore, all state variables still remain unchanged.

(3). $\Phi\,(e)$ is a linked counterpart of e in *Circus*. Therefore, the output pattern of communication, $c.e$, can be regarded as a synchronisation channel d which is equal to $c.e$. Then according to the proof above about the link rules for the synchronisation Channel, $(c.e(s_i,\cdots,s_j) \rightarrow \mathbf{Skip})$ has the same semantics as

$$(c.\Phi\,(e(s_i,\cdots,s_j))\,[\![s_i,\cdots,s_j/s_i,\cdots,s_j]\!] \rightarrow SKIP)$$

Finally, the rule is sound. □

Proof. $c.e \rightarrow A$ is the similar case to $c \rightarrow A$. □

Input Channel

Lemma 5.10.9 (Input Channel with Constrained Input)**.** *Link Rule 36 for prefixing with input channel is sound.*

Proof. (1). The constrained input prefixing $c?x : P \rightarrow A(x)$ in *Circus* is defined as a reactive design

$$c?x : P \rightarrow A(x) \mathrel{\widehat=} \mathbf{var}\ x \bullet \boldsymbol{R}(\mathbf{true} \vdash do_{\mathcal{I}}(c, x, P) \wedge v' = v) \mathbin{;} A(x)$$

[Oliveira [35, Definition B.15]]

where

$$do_{\mathcal{I}}(c, x, P) \mathrel{\widehat=} \left(\begin{array}{l} (tr' = tr) \wedge (\{e : \delta(c) \mid P \bullet (c, e)\} \cap ref' \neq \varnothing) \\ \lhd wait' \rhd \\ (tr' - tr \in \{e : \delta(c) \mid P \bullet \langle (c, e) \rangle\}) \wedge x' = snd(last(tr')) \end{array} \right)$$

[Oliveira [35, Definition B.14]]

- c is the channel name.
- $\delta(c)$ is the type of the channel c.
- P is a predicate which the value communicated on the channel c shall satisfy.

This semantics states that before the communication, it cannot refuse the communication of any value, which has the type $\delta(c)$ but satisfies P, on the channel c. After the engagement of the communication, the trace is extended by one event with the channel c and a value e, and the final value of x is exactly the same as the value communicated on the channel c. The behaviour will be like the action A with x replaced by e.

In addition, the input prefixing can be expressed as an external external choice in *Circus* as well.

$$c?x : P \rightarrow A(x) \mathrel{\widehat=} \Box_{x:\{e:\delta(c) \mid P\}}\ c.x \rightarrow A(x)$$

[Oliveira, Cavalcanti & Woodcock [73]]

provided $\{e : \delta(c) \mid P\}$ is finite.

(2). A constrained input prefixing $c?x : P \rightarrow A(x)$ in *Circus* is linked to a $CSP \parallel_B Z$ program

$$c?x : \{y \mid y{<}{-}\Phi(T_c), \Phi(P)\} \rightarrow \Phi\left(R_{wrt}\left(A(x)\right)\right)$$

by Link Rule 36.

In CSP, the prefixing with the constrained input $c?x : B \rightarrow P(x)$ denotes a guarded choice [24].

$$c?x : B \rightarrow P(x) = \Box_{x:B}\ c.x \rightarrow P(x)$$

Roscoe [[71]]

where B is a set.

Therefore,

$$\begin{array}{l} c?x : \{y \mid y{<}{-}\Phi(T_c), \Phi(P)\} \rightarrow \Phi\left(R_{wrt}\left(A(x)\right)\right) \\ = \Box_{\ x:\{y \mid y<-\Phi(T_c), \Phi(P)\}}\, c.x \rightarrow \Phi\left(R_{wrt}\left(A(x)\right)\right) \end{array}$$

(3). The set comprehension

$$\{y \mid y\text{<-}\Phi(T_c), \Phi(P)\}$$

actually is the linked counterpart of the set comprehension (according to the characteristic set comprehension in Table D.5)

$$\{e : \delta(c) \mid P\}$$

because $\Phi\left(T_c\right)$ and $\Phi\left(P\right)$ are the linked counterparts of T_c and P. Additionally, according to Link Rule 50, if the action A is a prefixed action AA, the rule is valid. Actually $c.x \rightarrow A(x)$ is a prefixed action and x will not evaluate state variables, therefore

$$\square_{x:\{y|y\text{<-}\Phi(T_c),\Phi(P)\}} c.x \rightarrow \Phi\left(R_{wrt}\left(A(x)\right)\right)$$

in CSP is a sound link of

$$\square_{x:\{e:\delta(c)|P\}}$$

from *Circus*.

Finally, the rule for constrained input prefixing is sound. □

Lemma 5.10.10 (Input Channel without Constrained Input)**.** *Link Rule 37 for prefixing with input channel is sound.*

Proof. For the simplified input prefixing $(c?x \rightarrow A(x))$, it is linked by Link Rule 37. It is equal to constrained input prefixing with the predicate $P = true$ [35, Definition B.16]. Therefore, the rule is sound. □

Multiple Data Transfer Channel

Lemma 5.10.11 (Multiple Data Transfer Channel)**.** *Link Rule 38 for prefixing with multiple-part communication is sound.*

Proof. For the channel with multiple inputs, or outputs, or the combination of them, the proof is very similar to that of the individual channel pattern. □

5.10.2.3 Guarded Action

Lemma 5.10.12 (Guarded Action)**.** *Link Rule 39 for guarded action is sound.*

Proof. (1). The semantics of guarded actions in *Circus* is defined as

$$(g) \mathbin{\&} A \mathrel{\widehat=} \boldsymbol{R}((g \Rightarrow \neg A^f_f) \vdash ((g \wedge A^t_f) \vee (\neg g \wedge tr' = tr \wedge wait')))$$

[Oliveira [35, Definition B.6]]

It states that if the guard condition g is $true$, it is the action A itself. Otherwise, it deadlocks as **Stop**.

(2). The guarded action in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\begin{cases} channel\ P\_Op\_s_i : \Phi(T_{s_i}) \\ \cdots \\ channel\ P\_Op\_s_j : \Phi(T_{s_j}) \\ HIDE\_CSPB = \{\!|P\_Op\_s_i, \cdots, P\_Op\_s_j|\!\} \\ P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \Phi\left(g\right) \ \& \ \Phi\left(R_{post}\left(A\right)\right) \end{cases}$$

by Link Rule 39.

As a $CSP \parallel B$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further.

$$
\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{c}
P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \\
\Phi(g) \ \& \ \Phi(R_{post}(A))
\end{array}\right) \\
\underset{ops}{{}_C\|_B} (B)_v^{v'}
\end{array}\right) \\
\setminus \{| Op\_s_i, \cdots, Op\_s_j |\} \\
\quad \exists v_0 \bullet \\
= \left(\begin{array}{l}
P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{c}
\Phi(g) \llbracket s_{gi}, \cdots, s_{gj} / s_{gi}, \cdots, s_{gj} \rrbracket \\
\& \ \Phi(R_{post}(A)) \llbracket s_{ai}, \cdots, s_{aj} / s_{ai}, \cdots, s_{aj} \rrbracket
\end{array}\right) \\
\underset{ops}{{}_C\|_B} (B)_{v_0}^{v'}
\end{array}\right)
\end{array}\right) \\
\quad \setminus \{| Op\_s_i, \cdots, Op\_s_j |\} \\
\quad \wedge_R (v = v_0) \\
\hfill [\text{Law 2.4.3}] \\
= \left(\begin{array}{l}
(\Phi(g) \llbracket s_{gi}, \cdots, s_{gj} / s_{gi}, \cdots, s_{gj} \rrbracket \ \& \ \Phi(R_{post}(A)) \llbracket s_{ai}, \cdots, s_{aj} / s_{ai}, \cdots, s_{aj} \rrbracket) \\
\underset{ops}{{}_C\|_B} (B)_v^{v'}
\end{array}\right) \\
\hfill [\text{Hiding [7, Section 3.5.1, L5]}]
\end{array}
$$

The $b \ \& \ P$ construct in CSP is equal to

$$b \ \& \ P \mathrel{\widehat=} \text{if } b \text{ then } P \text{ else } STOP$$

Therefore,

$$
\begin{array}{l}
\left(\left(\begin{array}{c}
\Phi(g) \llbracket s_{gi}, \cdots, s_{gj} / s_{gi}, \cdots, s_{gj} \rrbracket \\
\& \ \Phi(R_{post}(A)) \llbracket s_{ai}, \cdots, s_{aj} / s_{ai}, \cdots, s_{aj} \rrbracket
\end{array}\right) \underset{ops}{{}_C\|_B} (B)_v^{v'}\right) \\
= \begin{cases}
\left(\begin{array}{l}
\Phi(R_{post}(A)) \llbracket s_{ai}, \cdots, s_{aj} / s_{ai}, \cdots, s_{aj} \rrbracket \\
\underset{ops}{{}_C\|_B} (B)_v^{v'}
\end{array}\right) & \text{if } \Phi(g) \llbracket s_{gi} / s_{gi}, \cdots \rrbracket = true \\
STOP \ \underset{ops}{{}_C\|_B} (B)_v^{v'} & \text{if } \Phi(g) \llbracket s_{gi} / s_{gi}, \cdots \rrbracket = false
\end{cases} \\
= \begin{cases}
\left(\begin{array}{l}
\Phi(R_{post}(A)) \llbracket s_{ai}, \cdots, s_{aj} / s_{ai}, \cdots, s_{aj} \rrbracket \\
\underset{ops}{{}_C\|_B} (B)_v^{v'}
\end{array}\right) & \text{if } \Phi(g) \llbracket s_{gi} / s_{gi}, \cdots \rrbracket = true \\
STOP & \text{if } \Phi(g) \llbracket s_{gi} / s_{gi}, \cdots \rrbracket = false
\end{cases} \\
\hfill [\text{Law 2.4.1}]
\end{array}
$$

(3). $\Phi(g) \llbracket s_{gi}, \cdots, s_{gj} / s_{gi}, \cdots, s_{gj} \rrbracket$ is a linked counterpart of g. Since $\Phi(g)$ is a linked counterpart of g, the value of the guard condition to be evaluated in both *Circus* and $CSP \parallel_B Z$ is equal. In other words, if g is evaluated to be *true*, $\Phi(g)$ will be *true*. And if g is evaluated to be *false*, $\Phi(g)$ will be *false*. Furthermore, since $\Phi(R_{post}(A)) \llbracket s_{ai}, \cdots, s_{aj} / s_{ai}, \cdots, s_{aj} \rrbracket$ is a linked counterpart of the action A, and

$$\Phi(R_{post}(A)) \ \underset{ops}{{}_C\|_B} (B)_v^{v'}$$

behaves like A. Eventually, the semantics of the $CSP \parallel_B Z$ program in (2) is the same as that of the *Circus* guarded action in (1). The rule is sound. $\Box$

5.10.2.4 Sequential Composition

Lemma 5.10.13 (Sequential Composition of Actions)**.** *Link Rule 40 for sequential composition of actions is sound.*

Proof. (1). The semantics of the sequential composition in *Circus* is not defined as a reactive design and instead as a relational sequence in UTP.

$$A_1 \mathbin{;} A_2 \mathrel{\widehat=} \exists\, v_0 \bullet A_1[v_0/v'] \land A_2[v_0/v] \qquad \text{[Oliveira, Cavalcanti \& Woodcock [73]]}$$

(2). The sequential composition of actions in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{pre}\left(A_1\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(A_1\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(A_2\right)\right)\right)$$

by Link Rule 40.

As a $CSP \parallel B$ program, it is defined by Definition 2.4.1. In addition, when taking *HIDE_CSPB* into account, it is simplified further.

$$\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{c}P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow\\ \left(\Phi\left(R_{post}\left(A_1\right)\right) \mathbin{;} \Phi\left(R_{wrt}\left(A_2\right)\right)\right)\end{array}\right)\\ {}_{C}\|_{B}\ (B)_{v}^{v'}\\ \ ops\end{array}\right)\\ \setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\}\\ = \left(\left(\Phi\left(R_{post}\left(A_1\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!] \mathbin{;} \Phi\left(R_{wrt}\left(A_2\right)\right)\right)\ \underset{ops}{{}_{C}\|_{B}}\ (B)_{v}^{v'}\right)\end{array}$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

The semantics of the sequential composition is defined as a relational sequence as well.

$$P_1(v') \mathbin{;} P_2(v) \mathrel{\widehat=} \exists\, v_0 \bullet P_1(v_0) \land P_2(v_0) \qquad \text{[Cavalcanti \& Woodcock [68]]}$$

(3). Since $\Phi\left(R_{post}\left(A_1\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!]$ and $\Phi\left(R_{wrt}\left(A_2\right)\right)$ are the linked counterparts of actions A_1 and A_2 in *Circus*, then we can conclude the semantics of the linked sequential composition in CSP is the same as that in *Circus*. The rule is sound. □

5.10.2.5 External Choice

Lemma 5.10.14 (External Choice of Actions)**.** *Link Rule 41 for external choice of actions AA is sound.*

Proof. (1). The semantics of the external choice in *Circus* is defined as

$$A_1 \mathbin{\Box} A_2 = \boldsymbol{R}\left(\begin{array}{l}\left(\neg {A_1}_f^t \land \neg {A_2}_f^t\right)\\ \vdash\\ \left({A_1}_f^t \land {A_2}_f^t\right) \lhd tr' = tr \land wait' \rhd \left({A_1}_f^t \lor {A_2}_f^t\right)\end{array}\right)$$

[Oliveira [35, Definition B.7]]

The precondition states that the external choice will not diverge only if both actions will not diverge. The postcondition establishes that if the trace has not changed and the choice has not terminated, the behaviour of external choice is given by the conjunction of both actions; otherwise, the choice is made and the behaviour is either of actions. Additionally, it shows that state change will not resolve the choice because it is not constrained in the condition, and the choice is only made if an event of actions is executed or it has terminated. For example, in the following choice, $SExp_1$ and $SExp_2$ will not resolve the choice but c and d events will.

$$\left(SExp_1 \mathbin{;} c \rightarrow \mathbf{Skip}\right) \mathbin{\Box} \left(SExp_2 \mathbin{;} d \rightarrow \mathbf{Skip}\right)$$

(2). The external choice in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \to \left(\Phi\left(R_{post}\left(A_1\right)\right) \Box \Phi\left(R_{post}\left(A_2\right)\right)\right)$$

by Link Rule 41 where A_1 and A_2 should be AA.

As a $CSP \parallel_B Z$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further provided the initial construct of A_1 evaluates $s_{a1} = s_{a1i}, \cdots, s_{a1j}$ and the initial construct of A_2 evaluates $s_{a2} = s_{a2i}, \cdots, s_{a2j}$. All state variables evaluated in the initial construct of A_1 and A_2 are the union of s_{a1} and s_{a2}, denoted as $s_i, \cdots, s_j$. Then

$$\begin{aligned}
&\left(\begin{array}{l} \left(\ P\_Op\_s_i?s_i \to \cdots \to P\_Op\_s_j?s_j \to \left(\Phi\left(R_{post}\left(A_1\right)\right) \Box \Phi\left(R_{post}\left(A_2\right)\right)\right)\ \right) \\ {}_{C}\underset{ops}{\parallel}_B \left(B\right)_v^{v'} \end{array}\right) \\
&\setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\} \\
&= \left(\left(\begin{array}{l} \Phi\left(R_{post}\left(A_1\right)\right)[\![s_{a1i}, \cdots, s_{a1j}/s_{a1i}, \cdots, s_{a1j}]\!] \\ \Box \\ \Phi\left(R_{post}\left(A_2\right)\right)[\![s_{a2i}, \cdots, s_{a2j}/s_{a2i}, \cdots, s_{a2j}]\!] \end{array}\right) {}_{C}\underset{ops}{\parallel}_B \left(B\right)_v^{v'}\right)
\end{aligned}$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

The semantics of external choice in CSP is defined as

$$P_1 \Box P_2 = \boldsymbol{R}\left(\begin{array}{l} \left(\neg P1_f^t \wedge \neg P2_f^t\right) \\ \vdash \\ \left(P1_f^t \wedge P2_f^t\right) \lhd tr' = tr \wedge wait' \rhd \left(P1_f^t \vee P2_f^t\right) \end{array}\right)$$

[Cavalcanti & Woodcock [68, Law 137]]

(3). Because

$$\Phi\left(R_{post}\left(A_1\right)\right)[\![s_{a1i}, \cdots, s_{a1j}/s_{a1i}, \cdots, s_{a1j}]\!]$$

and

$$\Phi\left(R_{post}\left(A_2\right)\right)[\![s_{a2i}, \cdots, s_{a2j}/s_{a2i}, \cdots, s_{a2j}]\!]$$

are the linked counterparts of A_1 and A_2, the linked external choice in (2) has the same semantics as that in *Circus* in (1). Therefore, the rule is sound.

□

5.10.2.6 External Choice (Mutually Exclusive Guarded Actions)

Lemma 5.10.15 (External Choice of Actions (Mutually Exclusive Guarded Actions)). *Link Rule 41 for external choice of actions with mutually exclusive guards is sound.*

Proof. For external choice, if its two actions are guarded by mutually exclusive guards, its guarded actions do not need to be AA. If g_1 is evaluated to be true and consequently g_2 is false, then

$$\begin{aligned}
&\left(\left(g_1\right) \& A_1 \Box \left(g_2\right) \& A_2\right) \\
&= \left(\left(g_1\right) \& A_1 \Box \mathbf{Stop}\right) && \text{[Oliveira [35, Definition B.6]]} \\
&= \left(\left(g_1\right) \& A_1\right) && \text{[Oliveira [35, Law C.114]]}
\end{aligned}$$

Otherwise, if g_2 is evaluated to be true and consequently g_1 is false, then

$$\begin{aligned}
&\left(\left(g_1\right) \& A_1 \Box \left(g_2\right) \& A_2\right) \\
&= \left(\mathbf{Stop} \Box \left(g_2\right) \& A_2\right) && \text{[Oliveira [35, Definition B.6]]} \\
&= \left(\left(g_2\right) \& A_2\right) && \text{[Oliveira [35, Law C.114]]}
\end{aligned}$$

So the external choice does not provide its environment a choice of both actions. Actually, it just evaluates both guarded conditions and provides the environment with the guarded action. Therefore, it is not necessary to cope with the state changes in normal external choice because there is only one action available. In the end, the guarded actions are not required to be prefixed actions AA. □

5.10.2.7 Internal Choice

Lemma 5.10.16 (Internal Choice of Actions)**.** *Link Rule 42 for internal choice of actions is sound.*

Proof. (1). The semantics of the internal choice in *Circus* is not defined as a reactive design and instead simply as the disjunction of both actions.

$$A_1 \sqcap A_2 = A_1 \vee A_2 \qquad \text{[Oliveira [35, Definition B.8]]}$$

(2). The internal choice of actions in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(A_1\right)\right) \sqcap \Phi\left(R_{post}\left(A_2\right)\right)\right)$$

by Link Rule 42.

As a $CSP \parallel_B Z$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further provided the initial construct of A_1 evaluates $s_{a1} = s_{a1i}, \cdots, s_{a1j}$ and the initial construct of A_2 evaluates $s_{a2} = s_{a2i}, \cdots, s_{a2j}$. All state variables evaluated in the initial construct of A_1 and A_2 are the union of s_{a1} and s_{a2}, denoted as $s_i, \cdots, s_j$. Then

$$\begin{array}{l}\left(\begin{array}{l}\left(\; P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \left(\Phi\left(R_{post}\left(A_1\right)\right) \sqcap \Phi\left(R_{post}\left(A_2\right)\right)\right) \;\right) \\ {}_{C}\underset{ops}{\parallel}_{B}\left(B\right)_{v}^{v'}\end{array}\right) \\ \setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\} \\ = \left(\left(\begin{array}{l}\Phi\left(R_{post}\left(A_1\right)\right)[\![s_{a1i}, \cdots, s_{a1j}/s_{a1i}, \cdots, s_{a1j}]\!] \\ \sqcap \\ \Phi\left(R_{post}\left(A_2\right)\right)[\![s_{a2i}, \cdots, s_{a2j}/s_{a2i}, \cdots, s_{a2j}]\!]\end{array}\right) {}_{C}\underset{ops}{\parallel}_{B}\left(B\right)_{v}^{v'}\right) \\ \qquad\qquad \text{[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]}\end{array}$$

The semantics of the internal choice in CSP is defined as

$$P_1 \sqcap P_2 \mathrel{\widehat=} P_1 \vee P_2 \qquad \text{[Cavalcanti \& Woodcock [68]]}$$

(3). Because

$$\Phi\left(R_{post}\left(A_1\right)\right)[\![s_{a1i}, \cdots, s_{a1j}/s_{a1i}, \cdots, s_{a1j}]\!]$$

and

$$\Phi\left(R_{post}\left(A_2\right)\right)[\![s_{a2i}, \cdots, s_{a2j}/s_{a2i}, \cdots, s_{a2j}]\!]$$

are the linked counterparts of A_1 and A_2, the linked internal choice in (2) has the same semantics as that in *Circus* in (1). Therefore, the rule is sound. □

5.10.2.8 Parallel Composition (Disjoint Variables in Scope)

Lemma 5.10.17 (Parallel Composition (Disjoint Variables in Scope))**.** *Link Rule 43 for parallel composition of actions with disjoint variables in scope is sound.*

Proof. (1). In *Circus*, the semantics of the parallel composition is defined as

$$A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2 \mathrel{\widehat=}$$
$$\mathbf{R}\left(\begin{array}{l} \neg\exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} \left(A_1{}^f_f \,;\, (1.tr' = tr)\right) \wedge \\ \left(A_{2f} \,;\, (2.tr' = tr)\right) \wedge \\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array}\right) \\ \wedge \neg\exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} \left(A_{1f} \,;\, (1.tr' = tr)\right) \wedge \\ \left(A_2{}^f_f \,;\, (2.tr' = tr)\right) \wedge \\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array}\right) \\ \vdash \\ \left(\left(A_1{}^t_f \,;\, U1\,(out\alpha A_1)\right) \wedge \left(A_2{}^t_f \,;\, U2\,(out\alpha A_2)\right)\right)_{+\{v,tr\}} \,;\, M_{\parallel}(cs) \end{array}\right)$$

[Oliveira [35, Definition B.18]]

where

$$M_{\parallel}(cs) \mathrel{\widehat=} tr' - tr \in (1.tr - tr \parallel_{cs} 2.tr - tr) \wedge 1.tr \upharpoonright cs = 2.tr \upharpoonright cs$$
$$\wedge \left(\begin{array}{l} \left(\begin{array}{l} (1.wait \vee 2.wait) \wedge \\ ref' \subseteq ((1.ref \cup 2.ref) \cap cs) \cup ((1.ref \cap 2.ref) \setminus cs) \end{array}\right) \\ \lhd wait' \rhd \\ (\neg 1.wait \wedge \neg 2.wait \wedge MSt) \end{array}\right)$$

[Oliveira, Cavalcanti & Woodcock [73]]

$$MSt \mathrel{\widehat=} \forall\, v \bullet \left(\begin{array}{l} (v \in ns_1 \Rightarrow v' = 1.v) \wedge \\ (v \in ns_2 \Rightarrow v' = 2.v) \wedge \\ (v \notin ns_1 \cup ns_2 \Rightarrow v' = v) \end{array}\right)$$

[Oliveira, Cavalcanti & Woodcock [73]]

ns_1 and ns_2 are the partitions of the variables including state variables and local variables.

- The precondition states that if it is possible for either of A_1 and A_2 to diverge, the parallel composition will diverge. Otherwise, both A_1 and A_2 do not diverge, and the postcondition is the parallel by merge.

- The postcondition is defined as the parallel by merge technique [24]. To begin with, two actions run independently. And then the results are merged afterwards by $M_{\parallel}(cs)$.

  - It merges the trace first. The new extended trace is a merge of new events of two actions by the $\parallel_{cs}$ function [35, Definition B.18] with the restriction that the trace of both actions shall agree on events from cs.

  - If either of actions have not terminated, the parallel composition will not terminate, and refuses all events in cs that are refused by any actions and all events not in cs that are refused by both actions.

  - If either of actions have terminated, the parallel composition has terminated and the state is merged by MSt.

- According to MSt, both A_1 and A_2 can access variables in their scope, but only the variables in their partitions ns_1 and ns_2 can have an effect on the final value of these variables. The variables which are not in their partitions remain the same. In addition, it is impossible for one variable in both ns_1 and ns_2 because they are partitions of variables.

- According to the assumption of this rule

  $$\begin{aligned}&ns_1 = scpV(A_1)\\&ns_2 = scpV(A_2)\\&ns_1 \cap ns_2 = \varnothing\end{aligned}$$

  therefore from the state aspect, A_1 and A_2 access and update variables independently.

(2). The parallel composition of actions with disjoint variables in scope in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi(R_{mrg}(R_{pre}(A_1), R_{pre}(A_2))) \to \left(\begin{array}{l}\Phi(R_{post}(A_1))\\ \quad \underset{\Phi(cs)}{\parallel}\\ \Phi(R_{post}(A_2))\end{array}\right)$$

by Link Rule 43.

As a $CSP \parallel_B Z$ program, it is defined by Definition 2.4.1. In addition, when taking *HIDE_CSPB* into account, it is simplified further provided the initial construct of A_1 evaluates $s_{a1} = s_{a1i}, \cdots, s_{a1j}$ and the initial construct of A_2 evaluates $s_{a2} = s_{a2i}, \cdots, s_{a2j}$. All state variables evaluated in the initial construct of A_1 and A_2 are the union of s_{a1} and s_{a2}, denoted as $s_i, \cdots, s_j$. Then

$$\begin{aligned}&\left(\begin{array}{l}\left(P\_Op\_s_i?s_i \to \cdots \to P\_Op\_s_j?s_j \to \left(\begin{array}{l}\Phi(R_{post}(A_1))\\ \quad \underset{\Phi(cs)}{\parallel}\\ \Phi(R_{post}(A_2))\end{array}\right)\right)\\ {}_C\underset{ops}{\parallel}_B\,(B)_v^{v'}\end{array}\right)\\ &\setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\}\\ = &\left(\left(\begin{array}{l}\Phi(R_{post}(A_1))[\![s_{a1i}, \cdots, s_{a1j}/s_{a1i}, \cdots, s_{a1j}]\!]\\ \quad \underset{\Phi(cs)}{\parallel}\\ \Phi(R_{post}(A_2))[\![s_{a2i}, \cdots, s_{a2j}/s_{a2i}, \cdots, s_{a2j}]\!]\end{array}\right) {}_C\underset{ops}{\parallel}_B\,(B)_v^{v'}\right)\end{aligned}$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

Provided

$$\begin{aligned}&P_1 = \Phi(R_{post}(A_1))[\![s_{a1i}, \cdots, s_{a1j}/s_{a1i}, \cdots, s_{a1j}]\!]\\&P_2 = \Phi(R_{post}(A_2))[\![s_{a2i}, \cdots, s_{a2j}/s_{a2i}, \cdots, s_{a2j}]\!]\end{aligned}$$

then the parallel composition in CSP is defined by the parallel by merge in UTP.

$$\left(P_1 \underset{\Phi(cs)}{\parallel} P_2\right) \mathrel{\widehat=} ((P_1 ; U1(out\alpha P_1)) \wedge (P_2 ; U2(out\alpha P_2)))_{+\{v,tr\}} ; N$$

[Hoare & He [24, Paralle by Merge]]

where

$$N \mathrel{\widehat=} \left(\begin{array}{l}okay' = (P_1.okay \wedge P_2.okay) \wedge\\ wait' = (P_1.wait \vee P_2.wait) \wedge\\ ref' = ((P_1.ref \cup P_2.ref) \cap \Phi(cs)) \cup ((P_1.ref \cap P_2.ref) \setminus \Phi(cs)) \wedge\\ (tr' - tr) \in (P_1.tr - tr \parallel_{\Phi(cs)} P_2.tr - tr) \wedge\\ P_1.tr \upharpoonright \Phi(cs) = P_2.tr \upharpoonright \Phi(cs)\end{array}\right)$$
$$; SKIP$$

- If one of actions (A_1 or A_2) diverges, namely either $A_1.okay$ or $A_2.okay$ is false, thus the parallel composition diverges because $okay' = (A_1.okay \wedge A_2.okay) = false$.
- If not both of A_1 and A_2 have terminated, that is to say at least one of $A_1.wait$ and $A_2.wait$ is $true$, the parallel composition has not terminated because $wait' = true$. In this case, it refuses all events in cs that are refused by any actions and all events not in cs that are refused by both actions. If both of A_1 and A_2 have terminated, that is to say both $A_1.wait$ and $A_2.wait$ are $false$, the parallel composition has terminated because of $wait' = false$.
- The trace is merged by the operator, $\|_{\Phi(cs)}$, defined in [71]. It is the same as $\|_{cs}$ in [35, Definition B.18].
- From the state aspect, P_1 and P_2 update variables independently.

(3). Because

$$\Phi\left(R_{post}\left(A_1\right)\right)\llbracket s_{a1i}, \cdots, s_{a1j} / s_{a1i}, \cdots, s_{a1j}\rrbracket$$

and

$$\Phi\left(R_{post}\left(A_2\right)\right)\llbracket s_{a2i}, \cdots, s_{a2j} / s_{a2i}, \cdots, s_{a2j}\rrbracket$$

are the linked counterparts of A_1 and A_2, as a result, the linked $CSP \parallel_B Z$ program in (2) has the same semantics as the parallel composition of actions in *Circus* from both behavioural and state aspects. Therefore, the rule is sound. □

5.10.2.9 Parallel Composition (Disjoint Variables in Updating)

Lemma 5.10.18 (Parallel Composition (Disjoint Variables in Updating)). *Link Rule 44 for parallel composition of actions with disjoint variables in updating is sound.*

Proof. This rule has the different condition. Its assumption

$$\begin{aligned}
&wrtV\left(A_1\right) = ns_1\\
&wrtV\left(A_2\right) = ns_2\\
&wrtV\left(A_1\right) \cap scpV\left(A_2\right) = \varnothing\\
&wrtV\left(A_2\right) \cap scpV\left(A_1\right) = \varnothing
\end{aligned}$$

states that 1) all variables to be written in A_1 and A_2 are in its own partition and consequentially it is not necessary to discard any variables after termination, and 2) all variables to be written in one action are not seen by another action which makes it safe to update these variables.

The behaviour of the parallel composition (Disjoint Variables in Updating) is the same as the reasoning of the parallel composition (Disjoint Variables in Scope) above. Since the precondition of this rule establishes that the update of each variable in one action will not have impact on another action, and all these updates will not be discarded. From the state aspect, it is the same as the reasoning of the parallel composition above too. Finally, this rule is sound. □

5.10.2.10 Interleaving (Disjoint Variables in Scope)

Lemma 5.10.19 (Interleaving (Disjoint Variables in Updating)). *Link Rule 45 for interleaving of actions with disjoint variables in scope is sound.*

Proof. Since an interleaving is equal to a parallel composition on an empty synchronization channel for both *Circus* and CSP,

$$A_1 \;|||[\, ns_1 \mid ns_2 \,]|||\; A_2 = A_1 \;[\![\, ns_1 \mid \varnothing \mid ns_2 \,]\!]\; A_2 \qquad \text{[Oliveira [35, Law C.98]]}$$
$$P1 \underset{\varnothing}{\|} P2 = P1 \;|||\; P2 \qquad \text{Roscoe [[71]]}$$

according to the soundness of parallel composition in Section 5.10.2.8, then the rule is sound. □

5.10.2.11 Interleaving (Disjoint Variables in Updating)

Lemma 5.10.20 (Interleaving (Disjoint Variables in Updating))**.** *Link Rule 46 for interleaving of actions with disjoint variables in updating is sound.*

Proof. Since an interleaving is equal to a parallel composition on an empty synchronization channel for both *Circus* and CSP,

$$A_1 \;|||[\, ns_1 \mid ns_2 \,]|||\; A_2 = A_1 \;[\![\, ns_1 \mid \varnothing \mid ns_2 \,]\!]\; A_2 \qquad \text{[Oliveira [35, Law C.98]]}$$
$$P1 \underset{\varnothing}{\|} P2 = P1 \;|||\; P2 \qquad \text{Roscoe [[71]]}$$

according to the soundness of parallel composition in Section 5.10.2.9, then the rule is sound. □

5.10.2.12 Hiding

Lemma 5.10.21 (Hiding of Action)**.** *Link Rule 47 for hiding of action is sound.*

Proof. (1). The semantics of hiding in *Circus* is defined as

$$A \setminus cs \mathrel{\widehat=} \boldsymbol{R}\left(\exists\, s \bullet \left(\begin{array}{l} A[s, cs \cup ref'/tr', ref'] \wedge \\ (tr' - tr) = (s - tr) \upharpoonright (EVENT - cs) \end{array}\right)\right) ; \mathbf{Skip}$$

[Oliveira [35, Definition B.20]]

where $EVENT$ denotes a universal set of events. For the action A, it is the alphabet of A.

(2). The action hiding in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\,(R_{Pre}\,(A)) \rightarrow (\Phi\,(R_{Post}\,(A)) \setminus \Phi\,(cs))$$

by Link Rule 47.

As a $CSP \parallel B$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further.

$$\left(\begin{array}{l} \left(\begin{array}{c} P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \\ (\Phi\,(R_{Post}\,(A)) \setminus \Phi\,(cs)) \end{array}\right) \\ {}_C\underset{ops}{\|}_B\, (B)_v^{v'} \end{array}\right)$$
$$\setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\}$$
$$= \left((\Phi\,(R_{Post}\,(A))\,[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!] \setminus \Phi\,(cs)) \;\; {}_C\underset{ops}{\|}_B\, (B)_v^{v'}\right)$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

Provided $P = (\Phi\,(R_{Post}\,(A))\,[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!])$, the semantics of hiding in CSP is also defined as

$$P \setminus cs \mathrel{\widehat=} \boldsymbol{R}\left(\exists\, s \bullet \left(\begin{array}{l} P[s, cs \cup ref'/tr', ref'] \wedge \\ (tr' - tr) = (s - tr) \upharpoonright (\alpha P - cs) \end{array}\right)\right) ; SKIP$$

[Hoare & He [24, Definition 8.2.14]]

(3). Because $(\Phi\,(R_{Post}\,(A))\,[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!])$, $\Phi\,(cs)$, and $SKIP$ are the linked counterparts of A, cs and **Skip** in *Circus* respectively, the linked hiding in CSP has the same semantics as that in *Circus*. Therefore, the rule is sound. □

5.10.2.13 Recursion

Lemma 5.10.22 (Recursion of Action)**.** *Link Rule 48 for recursion of action is sound.*

Proof. (1). The semantics of the explicit recursion definition is defined as

$$\begin{aligned}&\mu\, X \bullet A(X)\\ &\mathrel{\widehat=} \textstyle\bigsqcap\{X \mid A(X) \sqsubseteq_{\mathcal{A}} X\} && \text{[Oliveira [35, Definition B.18]]}\\ &= \textstyle\bigsqcap\{X \mid [X \Rightarrow A(X)]\} && \text{[$\sqsubseteq_{\mathcal{A}}$ [35, Definition 3.1]]}\end{aligned}$$

which is the weakest fixed point of the equation $(X = A(X))$.

(2). The recursion in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\mathrm{let\ X} = \Phi\left(R_{wrt}\left(A(X)\right)\right)\ \mathrm{within\ X}$$

by Link Rule 48. This local definition defines a recursion, $X = \Phi\left(R_{wrt}\left(A(X)\right)\right)$ and its semantics is given by the weakest fixed point as well.

$$\textstyle\bigsqcap\{X \mid [X \Rightarrow \Phi\left(R_{wrt}\left(A(X)\right)\right)]\} \qquad \text{[Cavalcanti \& Woodcock [68]]}$$

(3). Because $\Phi\left(R_{wrt}\left(A(X)\right)\right)$ is the linked counterpart of $A(X)$, the local definition in (2) has the same semantics as the recursion in (1). The rule is sound. ☐

5.10.2.14 Iterated Sequential Composition

Lemma 5.10.23 (Iterated Sequential Composition of Actions)**.** *Link Rule 49 for iterated sequential composition of actions is sound.*

Proof. (1). The semantics of iterated operators is an expansion of the related operator in actions. For the sequential composition, its semantics is defined as

$$;\, x : T \bullet A(x) \mathrel{\widehat=} A(x_1)\,;\, A(x_2)\,;\cdots;\, A(x_n) \qquad \text{[Oliveira [35, Definition B.22]]}$$

provided T is a finite sequence and

$$T = \langle x_1, x_2, \cdots, x_n\rangle$$

(2). The iterated sequential composition in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{pre}\left(A(x)\right)\right) \to ;_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)$$

by Link Rule 49.

As a $CSP \parallel B$ program, it is defined by Definition 2.4.1. In addition, when taking *HIDE_CSPB* into account, it is simplified further.

$$\begin{aligned}&\left(\begin{array}{l}\left(\begin{array}{c}P\_Op\_s_i?s_i \to \cdots \to P\_Op\_s_j?s_j \to\\ \left(;_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)\right)\end{array}\right)\\ {}_{C}\underset{ops}{\parallel}_{B}\ (B)_v^{v'}\end{array}\right)\\ &\setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\}\\ &= \left(\left(;_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)[\![s_i, \cdots, s_j / s_i, \cdots, s_j]\!]\right)\ {}_{C}\underset{ops}{\parallel}_{B}\ (B)_v^{v'}\right)\end{aligned}$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

The iterated sequential composition in CSP is an expansion of the sequential composition as well.

$$
\begin{aligned}
&;_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\
&= \left(\begin{array}{l}
\Phi\left(R_{post}\left(A\left(\Phi\left(x_1\right)\right)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket ; \\
\Phi\left(R_{post}\left(A\left(\Phi\left(x_2\right)\right)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket ; \\
\ldots ; \\
\Phi\left(R_{post}\left(A\left(\Phi\left(x_n\right)\right)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket
\end{array}\right)
\end{aligned}
$$

(3). Since $\Phi\left(R_{post}\left(A\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket$ and $\Phi\left(x_i\right)$ are the linked counterparts of the action A and the expression x_i in *Circus*, and additionally the ; is the linked counterpart of ; in *Circus*, then we can conclude the semantics of the linked sequential composition in CSP is the same as that in *Circus*. The rule is sound. □

5.10.2.15 Iterated External Choice

Lemma 5.10.24 (Iterated External Choice of Actions)**.** *Link Rule 50 for iterated external choice of actions is sound.*

Proof. (1). The semantics of iterated operators is an expansion of the related operator in actions. For the external choice, its semantics is defined as

$$\Box\, x : T \bullet A(x) \mathrel{\widehat{=}} A(x_1) \Box A(x_2) \Box \cdots \Box A(x_n) \qquad \text{[Oliveira [35, Definition B.22]]}$$

provided T is a finite set

$$T = \{x_1, x_2, \cdots, x_n\}$$

(2). The iterated external choice in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \Box_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)$$

by Link Rule 50.

As a $CSP \parallel B$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further.

$$
\begin{aligned}
&\left(\begin{array}{l}
\left(\begin{array}{c}
P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \\
\Box_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)
\end{array}\right) \\
{}_{C}\underset{ops}{\parallel}_{B} \left(B\right)_{v}^{v'}
\end{array}\right) \\
&\setminus \{\!| Op\_s_i, \cdots, Op\_s_j |\!\} \\
&= \left(\left(\Box_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket\right) {}_{C}\underset{ops}{\parallel}_{B} \left(B\right)_{v}^{v'}\right)
\end{aligned}
$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

The iterated external choice in CSP is an expansion of the external choice operation as well.

$$
\begin{aligned}
&\Box_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\
&= \left(\begin{array}{l}
\Phi\left(R_{post}\left(A\left(\Phi\left(x_1\right)\right)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \Box \\
\Phi\left(R_{post}\left(A\left(\Phi\left(x_2\right)\right)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \Box \\
\ldots \Box \\
\Phi\left(R_{post}\left(A\left(\Phi\left(x_n\right)\right)\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket
\end{array}\right)
\end{aligned}
$$

(3). Since $\Phi\left(R_{post}\left(A\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket$ and $\Phi\left(x_i\right)$ are the linked counterparts of the action A and the expression x_i in *Circus*, and additionally the □ is the linked counterpart of □ in *Circus*, then we can conclude the semantics of the linked external choice in CSP is the same as that in *Circus*. The rule is sound.

□

5.10.2.16 Iterated Internal Choice

Lemma 5.10.25 (Iterated Internal Choice of Actions)**.** *Link Rule 51 for iterated internal choice of actions is sound.*

Proof. (1). The semantics of iterated operators is an expansion of the related operator in actions. For the external choice, its semantics is defined as

$$\bigsqcap x : T \bullet A(x) \mathrel{\widehat=} A(x_1) \sqcap A(x_2) \sqcap \cdots \sqcap A(x_n) \qquad \text{[Oliveira [35, Definition B.22]]}$$

provided T is a finite set

$$T = \{x_1, x_2, \cdots, x_n\}$$

(2). The iterated external choice in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \bigsqcap_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)$$

by Link Rule 51.

As a $CSP \parallel B$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further.

$$\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{c}P\_Op\_s_i?s_i \rightarrow \cdots \rightarrow P\_Op\_s_j?s_j \rightarrow \\ \bigsqcap_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)\end{array}\right) \\ {}_C\underset{ops}{\parallel}_B \left(B\right)_v^{v'}\end{array}\right) \\ \setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\} \\ = \left(\left(\bigsqcap_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!]\right) {}_C\underset{ops}{\parallel}_B \left(B\right)_v^{v'}\right)\end{array}$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

The iterated external choice in CSP is an expansion of the internal choice operation as well.

$$\begin{array}{l}\bigsqcap_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!] \\ = \left(\begin{array}{l}\Phi\left(R_{post}\left(A\left(\Phi\left(x_1\right)\right)\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!] \sqcap \\ \Phi\left(R_{post}\left(A\left(\Phi\left(x_2\right)\right)\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!] \sqcap \\ \ldots \sqcap \\ \Phi\left(R_{post}\left(A\left(\Phi\left(x_n\right)\right)\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!]\end{array}\right)\end{array}$$

(3). Since $\Phi\left(R_{post}\left(A\right)\right)[\![s_i, \cdots, s_j/s_i, \cdots, s_j]\!]$ and $\Phi\left(x_i\right)$ are the linked counterparts of the action A and the expression x_i in *Circus*, and additionally the $\sqcap$ is the linked counterpart of $\sqcap$ in *Circus*, then we can conclude the semantics of the linked internal choice in CSP is the same as that in *Circus*. The rule is sound. □

5.10.3 Command

5.10.3.1 Assignment

Lemma 5.10.26 (Assignment)**.** *Link Rule 52 for assignment is sound.*

Proof. (1). The semantics of assignments in *Circus* is defined as

$$\begin{array}{l}s_i, \cdots, s_j, l_k, \cdots, l_m := e_{s_i}, \cdots, e_{s_j}, e_{l_k}, \cdots, e_{l_m} \\ \mathrel{\widehat=} \mathbf{R}\left(\mathbf{true} \vdash \left(\begin{array}{l}tr' = tr \land \lnot wait' \land \\ s_i' = e_{s_i} \land \cdots \land s_j' = e_{s_j} \land \\ l_k' = e_{l_k} \land \cdots \land l_m' = e_{l_m} \land u' = u\end{array}\right)\right)\end{array}$$

[Oliveira [35, Definition B.31]]

It will not diverge, terminate successfully and leave the trace unchanged. In addition, it updates the variables in its RHS ($s_i, \cdots, s_j$ and $l_k, \cdots, l_m$) and leaves others ($u = v \setminus \{s_i, \ldots, s_j, l_k, \ldots, l_m\}$) remain the same. Furthermore, the assignment in *Circus* has an assumption that expressions in RHS of the assignment are defined.

(2). The assignment in *Circus* is linked to the *CSP* $\|_B$ *Z* program

$$\Omega_3 \left\{ \begin{array}{ll} \left(\begin{array}{l} P\_assOp \mathrel{\widehat=} \\ \left[\begin{array}{l} \Delta P\_StPar ; l_p? : T_{l_p} ; \cdots ; l_q? : T_{l_q}; \\ \quad l_k! : T_{l_k} ; \cdots ; l_m! : T_{l_m}; \\ \quad \Xi Q_1\_StPar ; \cdots \Xi Q_n\_StPar; \mid \\ P\_s_i' = e_{s_i}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \cdots \wedge \\ \quad P\_s_j' = e_{s_j}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \\ l_k! = e_{l_k}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \cdots \wedge \\ \quad l_m! = e_{l_m}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge u' = u \end{array} \right] \end{array} \right) & \text{State Part} \\ \left(\begin{array}{l} channel\ P\_assOp : \\ \quad \Phi\left(T_{l_p}\right) . \cdots . \Phi\left(T_{l_q}\right) . \Phi\left(T_{l_k}\right) . \cdots . \Phi\left(T_{l_m}\right) \\ HIDE\_CSPB = \{\!| P\_assOp |\!\} \\ (P\_assOp!l_p! \cdots !l_q?l_k? \cdots ?l_m \rightarrow SKIP) \end{array} \right) & \text{Behaviour Part} \end{array} \right.$$

[Ω_2 Rule 4]

by Link Rule 52.

According to Theorem 5.2.2, the semantics of this *CSP* $\|_B$ *Z* program is

$$\begin{array}{ll} \mathbf{R} \left(\begin{array}{l} (pre[l_o/ins?] \Rightarrow true) \\ \vdash \\ \left(\begin{array}{l} (pre[l_o/ins?] \wedge post[l_o, l_i/ins?, outs!] \wedge \neg wait' \wedge tr' = tr) \\ \vee \\ (\neg pre[l_o/ins?] \wedge tr' = tr \wedge wait') \end{array} \right) \end{array} \right) & \\ = \mathbf{R} \left(true \vdash \left(\begin{array}{l} P\_s_i' = e_{s_i}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \cdots \wedge \\ \quad P\_s_j' = e_{s_j}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \\ l_k' = e_{l_k}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \cdots \wedge \\ \quad l_m' = e_{l_m}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \\ \neg wait' \wedge tr' = tr \wedge u' = u \end{array} \right) \right) & [pre = true] \end{array}$$

where u denotes the variables apart from the variables ($s_i, \cdots, s_j$ and $l_k, \cdots, l_m$).

(3). Finally, compared to the semantics of the assignment in *Circus*, the linked program in *CSP* $\|_B$ *Z* updates the same state variables ($s_i, \cdots, s_j$) and local variables ($l_k, \cdots, l_m$) by the same corresponding expressions ($e_{s_i}, \cdots, e_{s_j}$ and $e_{l_k}, \cdots, e_{l_m}$). Furthermore, apart from these variables, all other variables u are left unchanged. Therefore, the semantics of assignments is the same as that of the linked program and the rule is sound. □

5.10.3.2 Alternation

Lemma 5.10.27 (Alternation)**.** *Link Rule 53 for alternation is sound.*

Proof. (1). The semantics of alternations in *Circus* is defined as

$$\mathbf{if} \, [\!] \, i \bullet g_i \rightarrow A_i \, \mathbf{fi} \mathrel{\widehat=} \mathbf{R}\left(\left(\bigvee i \bullet g_i \right) \wedge \left(\bigwedge i \bullet g_i \Rightarrow \neg A^f_{i\,f} \right) \vdash \bigvee i \bullet \left(g_i \wedge A^t_{i\,f} \right) \right)$$

[Oliveira [35, Definition B.35]]

The precondition specifies the condition of divergence: all guarded conditions are **false**, or any action guarded by a **true** guard diverges. Otherwise, any action guarded by a **true** guard is chosen from execution. And if more than one guard is **true**, then the behaviour is an arbitrary choice of an action from these guarded actions, exactly like internal choice.

(2). The alternation in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \left(\begin{array}{ll} & \Phi(\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \mathbf{div} \\ \Box & \Phi(g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \Phi(R_{post}(A_1)) \\ \Box & \Phi(g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \;\&\; (\Phi(R_{post}(A_1)) \sqcap \Phi(R_{post}(A_2))) \\ \Box & \ldots \\ \Box & \Phi(\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \;\&\; \left(\begin{array}{ll} & \Phi(R_{post}(A_i)) \\ \sqcap & \Phi(R_{post}(A_j)) \\ \sqcap & \Phi(R_{post}(A_k)) \end{array}\right) \\ \Box & \ldots \\ \Box & \Phi(g_1 \wedge g_2 \wedge \cdots \wedge g_n) \;\&\; \left(\begin{array}{ll} & \Phi(R_{post}(A_1)) \\ \sqcap & \Phi(R_{post}(A_2)) \\ \sqcap & \ldots \\ \sqcap & \Phi(R_{post}(A_n)) \end{array}\right) \end{array}\right) \end{array}\right)$$

by Link Rule 53.

As a $CSP \parallel_B Z$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further.

$$\left(\begin{array}{l} \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \left(\begin{array}{ll} & \Phi(\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \mathbf{div} \\ \Box & \Phi(g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \Phi(R_{post}(A_1)) \\ \Box & \Phi(g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \;\&\; (\Phi(R_{post}(A_1)) \sqcap \Phi(R_{post}(A_2))) \\ \Box & \ldots \\ \Box & \Phi(\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \\ & \qquad \& \left(\begin{array}{ll} & \Phi(R_{post}(A_i)) \\ \sqcap & \Phi(R_{post}(A_j)) \\ \sqcap & \Phi(R_{post}(A_k)) \end{array}\right) \\ \Box & \ldots \\ \Box & \Phi(g_1 \wedge g_2 \wedge \cdots \wedge g_n) \;\&\; \left(\begin{array}{ll} & \Phi(R_{post}(A_1)) \\ \sqcap & \Phi(R_{post}(A_2)) \\ \sqcap & \ldots \\ \sqcap & \Phi(R_{post}(A_n)) \end{array}\right) \end{array}\right) \end{array}\right) \\ {}_{C}\!\parallel_{B}{}_{ops} \; (B)_v^{v'} \end{array}\right) \\ \setminus \{\!|Op\_s_i, \cdots, Op\_s_j|\!\}$$

Table 5.1: Comparison of Alternation and its Translation Specification

g_1	g_2	*Circus* Behaviour	$CSP \parallel_B Z$ Condition	$CSP \parallel_B Z$ Behaviour
true	true	$A_1 \sqcap A_2$	$\Phi\left(g_1 \wedge g_2\right)$	$\Phi\left(R_{post}\left(A_1\right)\right) \sqcap \Phi\left(R_{post}\left(A_2\right)\right)$
true	false	A_1	$\Phi\left(g_1 \wedge \neg g_2\right)$	$\Phi\left(R_{post}\left(A_1\right)\right)$
false	true	A_2	$\Phi\left(\neg g_1 \wedge g_2\right)$	$\Phi\left(R_{post}\left(A_2\right)\right)$
false	false	**Chaos**	$\Phi\left(\neg g_1 \wedge \neg g_2\right)$	**div**

$$
= \left(\begin{array}{l}
\left(\begin{array}{l}
\quad \Phi\left(\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \ \& \ \mathbf{div} \\
\Box \left(\begin{array}{l} \Phi\left(g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \quad \& \ \Phi\left(R_{post}\left(A_1\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \end{array}\right) \\
\Box \left(\begin{array}{l} \Phi\left(g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \quad \& \left(\begin{array}{l} \quad \Phi\left(R_{post}\left(A_1\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \sqcap \ \Phi\left(R_{post}\left(A_2\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \end{array}\right) \end{array}\right) \\
\Box \ \ldots \\
\Box \left(\begin{array}{l} \Phi\left(\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \quad \& \left(\begin{array}{l} \quad \Phi\left(R_{post}\left(A_i\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \sqcap \ \Phi\left(R_{post}\left(A_j\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \sqcap \ \Phi\left(R_{post}\left(A_k\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \end{array}\right) \end{array}\right) \\
\Box \ \ldots \\
\Box \left(\begin{array}{l} \Phi\left(g_1 \wedge g_2 \wedge \cdots \wedge g_n\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \quad \& \left(\begin{array}{l} \quad \Phi\left(R_{post}\left(A_1\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \sqcap \ \Phi\left(R_{post}\left(A_2\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\ \sqcap \ \cdots \\ \sqcap \ \Phi\left(R_{post}\left(A_n\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \end{array}\right) \end{array}\right)
\end{array}\right) \\
{}_{C}\underset{ops}{\parallel}_{B} \left(B\right)_{v}^{v'}
\end{array}\right)
$$

[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]

(3). Compared to the alternation in *Circus*, the linked counterpart in $CSP \parallel_B Z$ has the same semantics. This is demonstrated in Table 5.1. In the table, two guarded actions, which are guarded by g_1 and g_2 respectively, are taken into account.

Since

$$
\begin{array}{l}
\Phi\left(g_1 \wedge g_2\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket \\
\ldots
\end{array}
$$

and

$$
\Phi\left(R_{post}\left(A\right)\right) \llbracket s_i, \cdots, s_j / s_i, \cdots, s_j \rrbracket
$$

are linked counterparts of $(g_1 \wedge g_2, \ldots)$ and A in *Circus* to $CSP \parallel_B Z$, and **div** and $\sqcap$ correspond to **Chaos** and $\sqcap$ in *Circus*, from the table we can conclude that the semantics of the alternation and its linked program is the same for two guarded actions in the alternation. We can use induction to easily get that the semantics of the alternation and its linked program is the same for any number of guarded actions. Therefore, the rule is sound. □

5.10.3.3 Variable Block

Lemma 5.10.28 (Variable Block). *Link Rule 54 for variable block is sound.*

Proof. (1). The semantics of variable block is defined as variable declarations in UTP. The initial value of variable x is chosen from T but arbitrary non-deterministic. The scope of variable x is between **var** and **end**.

$$\begin{aligned}
&\mathbf{var}\, x : T \bullet A \\
&\mathrel{\widehat=} \mathbf{var}\ x : T \,;\, A \,;\, \mathbf{end}\ x : T && \text{[Oliveira, Cavalcanti \& Woodcock [72]]} \\
&= \textstyle\bigsqcap\{\mathbf{var}\ x \,;\, x := k \mid k \in T\} \,;\, A \,;\, \mathbf{end}\ x : T \\
& && \text{[Hoare \& He [24, Variable Declaration L4]]}
\end{aligned}$$

(2). The variable block in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{pre}\left(A\right)\right) \rightarrow \textstyle\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}(A)\right), \{x\}\right)$$

by Link Rule 54.

As a $CSP \parallel_B Z$ program, it is defined by Definition 2.4.1. In addition, when taking $HIDE\_CSPB$ into account, it is simplified further.

$$\begin{aligned}
&\left(\begin{array}{l}
\left(\begin{array}{l}
Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\
\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}(A)\right), \{x\}\right)
\end{array}\right) \\
{}_C\|_B \left(B\right)_v^{v'} \\
\quad {}_{ops}
\end{array}\right) \\
&\setminus \{\!| Op\_s_i, \cdots, Op\_s_j |\!\} \\
&= \left(\textstyle\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}(A)\right)[\![s_i, \cdots, s_j / s_i, \cdots, s_j]\!], \{x\}\right) \ {}_C\underset{ops}{\|}_B \left(B\right)_v^{v'}\right) \\
& \qquad\qquad\qquad\qquad \text{[Law 2.4.3 and Hiding [7, Section 3.5.1, L5]]}
\end{aligned}$$

(3) The replicated internal choice in (2) introduces a variable x whose initial value is arbitrarily chosen from $\Phi(T)$ and the variable x is only valid within the scope of memory model process $\Phi\left(R_{post}(A)\right)$. It is the same as the variable block.

Furthermore, the memory model of process $\Phi\left(R_{wrt}(A)\right)$ (Definition B.1.5) just maintains the variables x in additional *Mem* processes, and the update and access of each variable in x within $\Phi\left(R_{wrt}(A)\right)$ are performed by *set* and *get* events provided by the *Mem* processes. According to Theorem C.1.28, $F_{Mem}(\Phi\left(R_{wrt}(A)\right), \{x\})$ has the same effect on variables x and the same behaviour as A in *Circus*. Additionally,

$$\Phi\left(R_{post}(A)\right)[\![s_i, \cdots, s_j / s_i, \cdots, s_j]\!]$$

in $CSP \parallel_B Z$ is equal to $\Phi\left(R_{wrt}(A)\right)$. Therefore, the variable block in (1) has the same semantics as the linked counterpart in (2). Then the rule is sound. □

5.10.3.4 Specification Statement

Lemma 5.10.29 (Specification Statement)**.** *Link Rule 55 for specification statement is sound.*

Proof. (1). The semantics of specification statements in *Circus* is defined as

$$w : [\, pre, post \,] \mathrel{\widehat=} \mathbf{R}(pre \vdash post \wedge \neg wait' \wedge tr' = tr \wedge u' = u)$$

[Oliveira [35, Definition B.32]]

(2). The specification statement in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\left\{\begin{array}{l}
\Omega_3 \left(\begin{array}{l}
P\_specOp \mathrel{\widehat=} \left[\begin{array}{l} \Delta P\_StPar \,;\, \Xi Q_1\_StPar \,;\, \cdots \,;\, \Xi Q_n\_StPar; \\ l_b? : T_{l_b} \,;\, l_a! : T_{l_a} \mid \\ (pre[l_b?/l_b] \land \exists\, u' : T_u \bullet post[l_b?, l_a!/l_b, l_a']) \\ \land\, s_u' = s_u \end{array}\right] \\[2ex]
P\_specOp\_fOp \mathrel{\widehat=} \left[\begin{array}{l} \Xi P\_StPar \,;\, \Xi Q_1\_StPar \,;\, \cdots \,;\, \Xi Q_n\_StPar; \\ l_b? : T_{l_b} \mid \neg\, \mathbf{pre}\, P\_specOp \end{array}\right]
\end{array}\right) \\[2ex]
\left(\begin{array}{l}
channel\ \ P\_specOp : \Phi\,(T_{l_b})\,.\Phi\,(T_{l_a}) \\
channel\ \ P\_fspecOp : \Phi\,(T_{l_b}) \\
HIDE\_CSPB = \{\!| P\_specOp, P\_fspecOp |\!\} \\
P\_specOp!l_b?l_a \rightarrow SKIP \,\Box\, P\_specOp\_fOp!l_b \rightarrow \mathbf{div}
\end{array}\right)
\end{array}\right.$$

by Link Rule 55.

According to Theorem 5.2.1, its semantics is

$$\begin{array}{l}
\boldsymbol{R}\left(pre \vdash \left(\exists\, u' : T_u \bullet post\right) \land \neg wait' \land tr' = tr\right) \\
= \boldsymbol{R}\left(pre \vdash post \land \neg wait' \land tr' = tr \land u' = u\right) \\
\hfill [u' \text{ is localised by } \exists \text{ and they are not able to be changed}]
\end{array}$$

(3). Therefore, we conclude the linked $CSP \parallel_B Z$ program in (2) has the same semantics to the original *Circus* specification statement. Hence the rule is sound. □

5.10.3.5 Assumption

Lemma 5.10.30 (Assumption). *Link Rule 56 for assumption is sound.*

Proof. The assumption $\{\, pre \,\}$ is simply syntactic sugar for the specification statement: $:[pre, true]$. Its proof is similar to the specification statement. For brevity, it is omitted. □

5.10.3.6 Coercion

Lemma 5.10.31 (Coercion). *Link Rule 57 for coercion is sound.*

Proof. The coercion $[\, post \,]$ is simply syntactic sugar for the specification statement in which its precondition always holds: $:[true, post]$. Its proof is similar to the specification statement. For brevity, it is omitted. □

5.10.3.7 Parametrisation By Value

Lemma 5.10.32 (Parametrisation By Value). *Link Rule 58 for parametrisation by value is sound.*

Proof. The parametrisation by value is defined in terms of the variable block, the assignment and the sequential composition.

$$(\mathbf{val}\, x : T \bullet A)\,(e) \mathrel{\widehat=} \mathbf{var}\, x : T \bullet x := e \,;\, A \qquad \text{[Oliveira [35, Definition B.37]]}$$

where $x \notin FV\,(e)$.

Therefore, its semantics can be derived from the semantics of these constructs. Because its link rule (Link Rule 58) is based on this transformation, the soundness of this rule relies on link rules for the variable block, the assignment and the sequential composition. According to the soundness of these constructs in Section 5.10.3.3, Section 5.10.3.1 and Section 5.10.2.4, this rule is sound. □

5.10.3.8 Parametrisation By Result

Lemma 5.10.33 (Parametrisation By Result). *Link Rule 59 for parametrisation by result is sound.*

Proof. Similarly, the parametrisation by result is defined in terms of the variable block, the assignment and the sequential composition as well.

$$(\mathbf{res}\, x : T \bullet A)\,(y) \mathrel{\widehat=} \mathbf{var}\, x : T \bullet A\,;\, y := x \qquad \text{[Oliveira [35]]}$$

Therefore, its semantics can be derived from the semantics of these constructs. Because its link rule (Link Rule 59) is based on this transformation, the soundness of this rule relies on link rules for the variable block, the assignment and the sequential composition. According to the soundness of these constructs in Section 5.10.3.3, Section 5.10.3.1 and Section 5.10.2.4, this rule is sound. □

5.10.3.9 Parametrisation By Value-Result

Lemma 5.10.34 (Parametrisation By Value-Result). *Link Rule 60 for parametrisation by value-result is sound.*

Proof. Similarly, the parametrisation by value-result is defined in terms of the variable block, the assignment and the sequential composition as well.

$$(\mathbf{vres}\, x : T \bullet A)\,(y) \mathrel{\widehat=} \mathbf{var}\, x : T \bullet x := y\,;\, A\,;\, y := x \qquad \text{[Oliveira [35]]}$$

Therefore, its semantics can be derived from the semantics of these constructs. Because its link rule (Link Rule 60) is based on this transformation, the soundness of this rule relies on link rules for the variable block, the assignment and the sequential composition. According to the soundness of these constructs in Section 5.10.3.3, Section 5.10.3.1 and Section 5.10.2.4, this rule is sound. □

5.10.4 Renaming

Lemma 5.10.35 (Renaming). *Link Rule 61 for renaming is sound.*

Proof. (1). The semantics of renaming in *Circus* is just a syntactic substitution of the name of new variables for the old variables.

(2). The renaming of action in *Circus* is linked to the $CSP \parallel_B Z$ program by Link Rule 61. It is just a syntactic substitution of variables.

(3). Therefore, the rule is sound. □

5.10.5 Action Invocation

Lemma 5.10.36 (Action Invocation). *Link Rule 62 for action invocation is sound.*

Proof. (1). The semantics of action invocations, namely reference to actions, is defined by the copy rule: it is the body of action.

(2). The action invocation in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\,(R_{wrt}\,(B(A)))$$

by Link Rule 62.

(3). According to Definition B.2.15, $B(A)$ denotes the body of the action A. Additionally, $\Phi\,(R_{wrt}\,(B(A)))$ in CSP is the link counterpart of $B(A)$ in *Circus*. Thus the rule is sound. □

5.10.6 Parametrised Action

5.10.6.1 Parametrised Action

Lemma 5.10.37 (Parametrised Action)**.** *Link Rule 65 for parametrised action is sound.*

Proof. (1). The semantics of a parametrised action invocation $A(e)$ is defined as an action invocation with the parameters substituted by expressions e.

$$A(e) \mathrel{\widehat=} B(A)[e/x] \qquad \text{[Oliveira [35, Definition B.29]]}$$

provided

$$A \mathrel{\widehat=} x : T \bullet B(A)$$

(2). The parametrised action $A(e)$ in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{wrt}\left(B(A)[e/x]\right)\right)$$

by Link Rule 65.

(3). Because $\Phi\left(R_{wrt}\left(B(A)[e/x]\right)\right)$ in CSP is the linked counterpart of $B(A)[e/x]$ in *Circus*, the semantics of the linked counterpart is the same as that of the parametrised action. Therefore, the rule is sound. □

5.10.6.2 Unnamed Parametrised Action

Lemma 5.10.38 (Unnamed Parametrised Action)**.** *Link Rule 64 for unnamed parametrised action is sound.*

Proof. (1). The semantics of unnamed parametrised action is defined as with the parameter substituted by expression e.

$$(x : T \bullet A)(e) \mathrel{\widehat=} A[e/x] \qquad \text{[Oliveira [35, Definition B.29]]}$$

(2). The unnamed parametrised action in *Circus* is linked to the $CSP \parallel_B Z$ program

$$\Phi\left(R_{wrt}\left(A[e/x]\right)\right)$$

by Link Rule 64.

(3). Because $\Phi\left(R_{wrt}\left(A[e/x]\right)\right)$ in CSP is the linked counterpart of $A[e/x]$ in *Circus*, the semantics of the linked counterpart is the same as that of the unnamed parametrised action. Therefore, the rule is sound. □

5.11 Summary

This chapter gives the soundness of our link from *Circus* to $CSP \parallel_B Z$. The reason to exclude Ω_3, which translates Z in ZRM to B, is because the Ω_3 relies on the translator in ProB. In order to prove each link rule in Appendix E is sound, our strategy is to give the UTP semantics to the original *Circus* construct, then give the UTP semantics to the linked counterpart in $CSP \parallel_B Z$, and finally compare their semantics. If they have the same semantics, then we say the rule is sound.

Then two important theorems (Theorem 5.2.1 and Theorem 5.2.2 about the semantics of the linked counterpart of schema expressions) are given. Finally, we give the soundness to each construct in *Circus*.

Chapter 6

Translator

To facilitate the translation of *Circus* models to the final resultant $CSP \parallel_B Z$ according to the link defined in Chapter 4, a translator is developed. In this chapter, we present our considerations of the translator development, the procedure of the translation, a number of challenges and the corresponding solutions, and finally the limitations. The source code information is given in Appendix F.

6.1 Overall Translation Procedure

Basically, the *Circus* to $CSP \parallel_B Z$ translator, namely *Circus2ZCSP*, is supposed to translate a *Circus* model in Unicode or the LaTeX markup to a model in $CSP \parallel_B Z$. And a $CSP \parallel_B Z$ model actually consists of two files: one Z specification and one CSP specification. In order to manipulate constructs of a *Circus* model by our link definition, the first step of the translator is to parse and typecheck the model to find syntax and type errors, and sequentially turn the model in Unicode or LaTeX into an internal abstract syntax tree (AST) if there is no error detected. Then the translator manipulates the internal AST and converts it into internal representations of the target Z model in ZRM and the target CSP model. In the end, these internal Z and CSP models are output from the translator, which results in a final Z model in LaTeX and a CSP model in CSP_M. Then they are able to be model-checked by ProB.

The overall translation procedure is illustrated in detail in Figure 6.1. Since the parser and typechecker for *Circus* have been integrated into the CZT project, in order to reuse them, our translator is written in Java and based on CZT as well. In the figure, we use different node shapes to denote input and output files (rectangle with shadow), existing modules in CZT (rectangle with round corners), and newly developed modules (rectangle with round corners and double lines). Each step is described as follows.

- Markups of input *Circus* models, that the translator can support, rely on the parser and typechecker in CZT. The LaTeX markup is normally used for the input model.

- Then the *Circus* parser is responsible for the generation of AST from the input LaTeX model.

- Afterwards, the *Circus* typechecker checks type, scope or naming errors in the AST according to type inference rules, and additionally annotates that AST with section and type environments. The new AST is abbreviated to AST_+.

- The new R_{wrt} module rewrites an AST_+, according to our rewrite rules defined in Section 4.3, to an AST_{+R}, a rewritten AST_+. The internal implementation of the R_{wrt} module is displayed in Section 6.2.

- After the rewrite, the AST_{+R} is translated to Z and CSP by Ω and Φ separately.

- For the Φ function, it generates the final CSP model from AST_{+R}. Because CZT is not capable of parsing and typechecking CSP, the output of Φ is a CSP model ("a_csp.csp") directly. The detail of the Φ module is given in Section 6.3.
- For the Ω function,
  - it merges the state part of *Circus* from the input AST_{+R} to form an ASTZ which is an AST for a Z model in ISO Standard Z dialect,
  - this ASTZ is converted to its corresponding LaTeX markup by the Z Pretty Printer in CZT,
  - then the LaTeX markup is fed to the Z Parser and TypeChecker in CZT to check if the translated Z model is correct and free of errors, and at the same time an annotated AST, ASTZ_{+}, is generated if no error is found,
  - finally the ASTZ_{+} along with the LaTeX markup of the Z model from the Pretty Printer are input to Ω_2 which transforms them into a Z model in ZRM ("a_z.tex").
  - The Ω modules are described in Section 6.4.
- Eventually, a resultant $CSP \parallel_B Z$ model, which is composed of one CSP model and one Z model, is obtained.

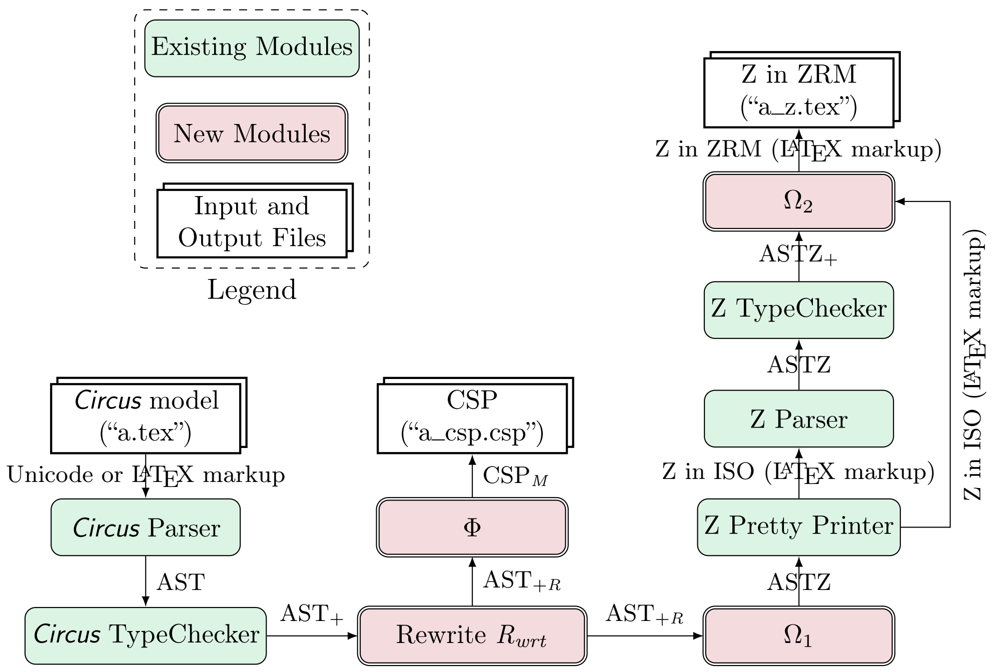

Figure 6.1: Overall Translation Procedure

6.2 Rewrite (R_{wrt}) Module

The rewrite procedure is shown in Figure 6.2. In addition to implementation of the R_{wrt} rules defined in Section 4.3, the rewrite module R_{wrt} has other practical considerations, such as identifier naming pattern checking, global schemas localisation, and schemas as predicates to predicates in the behavioural part. These are discussed as follows.

The input to the rewrite module is AST_{+} from the *Circus* typechecker. Since inheriting sections are new to ISO Z but not in ZRM Z, our first step of rewrite is to combine them together into one big section.

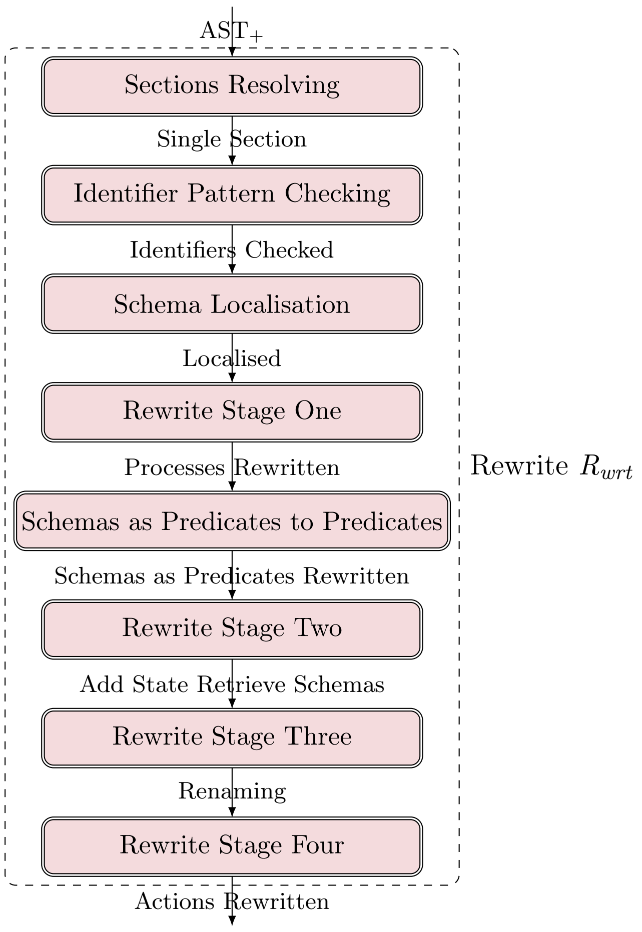

Figure 6.2: Rewrite Procedure

6.2.1 Sections Resolving

According to inheriting sections in ISO Z, one section and its parent sections might refer to the same sections. In order to include them all into a big section in the right order, an algorithm is designed to include one section's parent sections and is described as follows.

Step 1, declare a String stack, *section_stack_*, to denote visited parent sections, and a map from section names to their paragraph lists, *sections_paralist_map_*

Step 2, get a list of one section's parent sections. If one section's parent section is one of standard toolkits, then just skip it since the standard toolkits have their counterparts in ZRM. Otherwise, the parent section is added to the list. The standard toolkits include

- `"standard_toolkit"`
- `"number_toolkit"`
- `"set_toolkit"`
- `"relation_toolkit"`
- `"function_toolkit"`
- `"sequence_toolkit"`

- `"circus_toolkit"`

Step 3, check the parent sections list and if one section's parent sections are empty or all have been visited (all in *section_stack_*), then just push the current section's name to *section_stack_* and add its paragraphs list to *sections_paralist_map_*. Then go to Step 7.

Step 4, for each parent section, if it has been visited (its name in *section_stack_*), continue to check next parent section in the list.

Step 5, otherwise, if this parent section has not been visited, mark this parent section as the current section and repeat Step 2 to Step 6 to recursively include its parent section's parent sections. Finally include this parent section's paragraph list into *sections_paralist_map_*.

Step 6, push the current section's name to *section_stack_* and add its paragraphs list to *sections_paralist_map_*.

Step 7, all sections have now been visited. Create a new empty paragraph list *paralist*.

Step 8, for each section in *section_stack_*, according to its order, get its corresponding paragraph list from *sections_paralist_map_*, and then add it to *paralist*.

Step 9, finally *paralist* is the new paragraphs list that includes all paragraphs from the section and all its parent sections. The final step just simply replaces the paragraphs list of the section by this new big paragraphs list *paralist*.

6.2.2 Identifier Pattern Checking

According to Section 4.2.1, the identifier naming pattern allowed in our solution is restricted, therefore this step is to check whether all names in the model comply with the pattern. If there are any names that are not allowed,the translator quits with an error to show these names.

6.2.3 Schema Localisation

Because Ω_1 Rule 1 needs to merge Z parts from multiple processes into one Z model, R_{wrt} Rule 24 is defined in the early stage to rename all state components, schemas and actions within processes. However, an issue arises due to renaming. If a schema is defined globally—it is not within a process, it might be referred to in more than two processes. Consequently, in one process its declared variables and the schema name are renamed by prefixing with the process's name. Then in another process its declared variables and the schema name are renamed by prefixing with this process's name. Since the names of two processes are different, it is impossible for a schema to be renamed to two different names. In order to address this issue, our solution is to make a copy of the schema in the processes that refer to the schema. Finally, if a global schema is not referred globally, it is removed since it has been duplicated within the processes.

The algorithm is described below. Firstly, we define *Node* to be a pair from schema name to process name

$$Node = String \times String$$

and *Entry* to be a quadruple from the node, to a set of nodes that the node refers to, global reference number, and local reference number.

$$Entry = Node \times \mathbb{P}\, Node \times \mathbb{N} \times \mathbb{N}$$

In this module, we use a map ($\mathit{ref\_map\_} : \mathit{Node} \pinj \mathit{Entry}$) to keep cross-references of all global schema definitions. The global axiomatic definitions are excluded because they are always global. The detailed algorithm is presented as follows.

Step 1, for each global definition, repeat Step 2 to Step 4.

Step 2, for the paragraph of a global definition, traverse its body to get a set of nodes (*setNodes*) that the paragraph refers to.

Step 3, then for each node in this set, if they are schemas, then get its corresponding entry from the map *ref_map_* and increment its global reference number by one.

Step 4, if this global definition is a schema, then create a new one

$$(\mathit{Node}(\mathit{pname}, \mathit{null}), \mathit{Entry}\,(\mathit{Node}(\mathit{pname}, \mathit{null}), \mathit{setNodes}, 0, 0))$$

in the map, where *pname* denotes the paragraph name and *null* means it is global.

Step 5, all global schemas now have their reference information in the map. Now start to visit all basic processes.

Step 6, for each basic process, traverse all its paragraphs. If a reference to a global schema is found, then copy this schema to the beginning of this process and increment this schema's local reference number in the map by one. At the same time, copy all schemas, that are referred by this schema, to the beginning of this process and increment their local reference number by one as well.

Step 7, now start to remove all schemas of which the global reference number is 0.

Step 8, for each node in the map, if its global reference number is 0, then remove its corresponding paragraph from the section. At the same time, get a set of referred nodes in its entry and decrease their global reference number by one.

Step 9, if all nodes in the map have the global reference number larger than zero, then complete and quit this module. Otherwise, go back to Step 7.

Finally, all global schemas are copied into the processes if they are referred to in these processes, and some global schemas that are not referred to globally any more are removed.

An example is given below.

```
Add [(Unit, null)]: []<0, 0>
Update [(Unit, null)]: []<1, 0>
Add [(QSensor, null)]: [(Unit, null)]<0, 0>
Update [(QSensor, null)]: [(Unit, null)]<1, 0>
Add [(InitQSensor, null)]: [(QSensor, null)]<0, 0>
Update [(Unit, null)]: []<2, 0>
Add [(VSensor, null)]: [(Unit, null)]<0, 0>
Update [(Unit, null)]: []<3, 0>
Add [(Sensor, null)]: [(Unit, null)]<0, 0>
 ...
Decrease reference from (QSensor, null)
  Update [(Unit, null)]: []<2, 0>
Decrease reference from (Sensor, null)
  Update [(Unit, null)]: []<1, 0>
Decrease reference from (VSensor, null)
  Update [(Unit, null)]: []<0, 0>
```

In the first line `line#1`, a new node *Unit* is added to the map with no reference, and both global and local reference numbers equal to zero. Then in `line#2`, its global reference number is increased by one because it is referred to in *QSensor*. Sequentially, its global reference number becomes 2 (`line#6`) and 3 (`line#8`) because it is further referred to in *VSensor* and *Sensor*. Afterwards, in `line#11`, *QSensor* is going to be removed and then the global reference number of *Unit* is decreased by one and becomes 2 (`line#12`). It continues to be reduced to 1 (`line#14`) and 0 (`line#16`). Finally, *Unit* is going to be removed since it is not referred to globally any more.

6.2.4 Rewrite Stage One

In this stage, global definitions, except basic processes, are rewritten by R_{wrt} Rule 4 to 22. It also includes the rewrite of generic constructs by R_{wrt} Rule 47 to 50.

6.2.5 Schemas as Predicates to Predicates

Schemas can be used in the behavioural part of *Circus* as predicates. $c.IsZero$, where the type of c is boolean and $IsZero$ is a schema reference, is an example. Unlike schemas as predicates in the Z part which can be evaluated by ProB easily, schemas as predicates in the CSP part have to be transformed into predicates to remove schemas.

In order to extract the predicate part from a schema, the direct way is through normalisation. However, the normalisation in CZT does not work as expected. For example, the schema

$$\left[AnalyserState \mid \left(\begin{array}{l} stops \geq 3 \lor DangerZone \lor \\ emergencyCond = 1 \lor transmissionFailure \in signals \end{array} \right) \right]$$

is normalised to

$$\left[\ldots \mid AnalyserState \land \left(\begin{array}{l} stops \geq 3 \lor DangerZone \lor \\ emergencyCond = 1 \lor transmissionFailure \in signals \end{array} \right) \right]$$

The declaration part is skipped and it is correct. However, the predicate part is obviously not what we expect since two schemas *AnalyserState* and *DangerZone* are not further expanded. Finally, we decide to implement a class, `PredicateListExpansionVisitor`, to extract the predicate part from a schema directly instead of the pattern match used in normalisation. The reason for having this step after *Rewrite Stage One* is that generic constructs have been resolved and this step does not need to cope with them.

Our solution is to traverse the declaration part and the predicate part of the schema separately to get their corresponding predicates, then the overall predicate of this schema is a conjunction of the predicate from the declaration part and the predicate from the predicate part. For example, the predicate of the schema

$$IsZero == [x : 0 \mathinner{..} 3 \mid x = 0]$$

is a conjunction of the predicate, $x \in 0 \mathinner{..} 3$, from $x : 0 \mathinner{..} 3$, and the predicate, $x = 0$, from $x = 0$.

$$x \in 0 \mathinner{..} 3 \land x = 0$$

For schema expressions as predicates, they are recursively expanded in this way. By this, negating an expression as a predicate would get the right result. For instance, the predicate of a negation expression $\neg IsZero$ where

$$IsZero == [[x : 0 \mathinner{..} 3 \mid x < 2] \mid x = 0]$$

becomes

$$\neg ((x \in 0 \mathrel{..} 3) \land (x < 2) \land (x = 0))$$

One inconvenience from our solution is duplicate constraints, from declaration parts, in the final predicate. An example below illustrates this inconvenience. According to our solution, the predicate of $\mathit{IsZero} \lor \mathit{NotZero}$ where

$$\mathit{IsZero} == [x : 0 \mathrel{..} 3 \mid x = 0]$$
$$\mathit{NotZero} == [x : 0 \mathrel{..} 3 \mid x \neq 0]$$

is

$$(x \in 0 \mathrel{..} 3 \land x = 0) \lor (x \in 0 \mathrel{..} 3 \land x \neq 0)$$

In the predicate, there are two duplicate $x \in 0 \mathrel{..} 3$ though we can manually rewrite to only one

$$x \in 0 \mathrel{..} 3 \land (x = 0 \lor x \neq 0)$$

Most significantly, duplicate predicates only make the final CSP model bigger but will not cause semantic problems. Therefore, we just keep duplicate predicates.

6.2.6 Rewrite Stage Two

This stage implements R_{wrt} Rule 23 by adding operation schemas in each explicitly defined process to retrieve the values of state components. The rule is very simple and clear. But one issue that arises from it is how to get a list of state components and their corresponding types. Though the state paragraph of a basic process is marked by the **state** keyword, the paragraph might be an abbreviation or a schema with complex schema expressions. Therefore, to get a list of state components in the state paragraph, it is necessary to expand its schema expressions to a schema of which all state variables are in the declaration part. However, normalisation is not expected. For instance, a state schema *State* below has a schema reference *StateA* in its declaration.

$$\mathit{StateA} == [\, a : 0 \mathrel{..} 3 \,]$$

$$\textbf{state}\ \ \mathit{State} == [\, \mathit{StateA} \, ; b : \{1, 3\} \mid a < b \,]$$

Normalisation will result in

$$\textbf{state}\ \ \mathit{State} == [\, a : \mathbb{A} \, ; b : \mathbb{A} \mid a \in 0 \mathrel{..} 3 \land b \in \{1, 3\} \land a < b \,]$$

Obviously the names of state components a and b are identified but the problem is their types. $\mathbb{A}$ is a carrier set, a set of all values in the number system, that makes it hard to animate and model-check a CSP specification when $\mathbb{A}$ is linked to CSP, compared to the restricted set $0 \mathrel{..} 3$. Therefore, it is better to identify the types of a and b as $0 \mathrel{..} 3$ and $\{1, 3\}$.

To accomplish this identification, we implement `DeclListExpansionVisitor`, a new class, in our translator to get a list of variables declared, including input and output variables, and their corresponding types.

Expansion of the declaration list by recursively traversing schema expressions without normalisation may cause problems, especially for schema negation expression. For the negation of a schema, $\neg \mathit{IsZero}$, where

$$\mathit{IsZero} == [x : 0 \mathrel{..} 3 \mid x = 0]$$

is equal to

$$\neg IsZero == [x : \mathbb{A} \mid \neg (x \in 0 \mathrel{..} 3 \land x = 0)]$$

by normalisation. Its declaration part only has a unique and canonical form. This is correct. However our solution with `DeclListExpansionVisitor` for $\neg IsZero$ will return the variable x and its type as a set $0 \mathrel{..} 3$, which is different from the normalisation solution.

By our solution, according to the state schema *State* given above, this rewrite stage will result in two additional operation schemas below.

$$\begin{aligned}&Op\_a == [\, \Xi State \,;\, a! : 0 \mathrel{..} 3 \mid a! = a \,]\\&Op\_b == [\, \Xi State \,;\, b! : \{1, 3\} \mid b! = b \,]\end{aligned}$$

Since the types of state components a and b are $0 \mathrel{..} 3$ and $\{1, 3\}$ respectively, Op_a and Op_b having their output variables $a!$ and $b!$ with the same types will not cause problems. When their corresponding schema expressions as actions are linked to CSP by Φ Rule 8, two channels are declared.

$$\begin{aligned}&\textbf{channel}\ \ Op\_a : \{0..3\}\\&\textbf{channel}\ \ Op\_b : \{1, 3\}\end{aligned}$$

Instead, the normalisation will result in two operation schemas

$$\begin{aligned}&Op\_a' == [\, \Xi State \,;\, a! : \mathbb{A} \mid a! = a \,]\\&Op\_b' == [\, \Xi State \,;\, b! : \mathbb{A} \mid b! = b \,]\end{aligned}$$

and their channel declaration will be

$$\begin{aligned}&\textbf{channel}\ \ Op\_a : Int\\&\textbf{channel}\ \ Op\_b : Int\end{aligned}$$

If the type of a state component is complicated such as $\mathbb{P}(\mathbb{A} \times \mathbb{A} \times \mathbb{A})$, it is hard to load and animate in ProB. However, our solution keeps types of state components as small as possible. Comparatively, it is easier to model-check and animate the CSP model.

6.2.7 Rewrite Stage Three

This stage simply renames all state components, schemas, and actions within a basic process by prefixing the name of this process according to R_{wrt} Rule 24.

6.2.8 Rewrite Stage Four

In this stage, the main action of each basic process is rewritten by R_{wrt} Rule 25 to 46. After this stage,

- all accesses of state components in actions are through their corresponding schema expressions,
- all implicit recursions have been made explicitly,
- action definitions, except the main action, are removed because their references in the main action have been rewritten by the action invocation rule, R_{wrt} Rule 35.

6.3 Φ Module

According to Figure 6.1, after the rewrite stage, the rewritten *Circus* AST_{+R} is translated to the final CSP by Φ. Since most of constructs in *Circus* are linked to Z according to our solution, it is not necessary to translate all these constructs in *Circus* to CSP. The translation of the full *Circus* model to CSP may cause additional problems as well because some constructs might not be supported. Therefore, the best way is to translate only those constructs that are needed in the final CSP.

In order to overcome this problem, we introduce a similar cross-reference mechanism in *Schema Localisation*. A map ($refmap\_ : Node \nrightarrow \mathbb{P}\, Node$), from a node to a set of nodes that it refers to, is introduced in the beginning of this module. We recursively traverse AST_{+R} to build up this map. For example, for a free type

$$SState ::= sokay \mid sfailed$$

three entries are added to the map

```
Put [sokay, null]: [(SState, null) ]
Put [sfailed, null]: [(SState, null) ]
Put [SState, null]: []
```

These entries in the map mean that if *sokay* or *sfailed* is used in CSP, it has to include the *SState* definition which would not include further definitions. And another example is the channel declaration

$$\textbf{channel}\ pumps : (PumpIndex \rightarrow InputPState) \times VAction$$

the entries listed below are added to the map

```
Put [PumpIndex, null]: []
Put [VAction, null]: []
Put [popen, null]: [(PState, null) ]
Put [pclosed, null]: [(PState, null) ]
Put [InputPState, null]: [(pclosed, null) (popen, null) ]
Put [pumps, null]: [(PumpIndex, null) (VAction, null)
                    (InputPState, null) ]
```

They denote that if the *pumps* channel is used in CSP, it has to include the definition of *PumpIndex*, *VAction*, and *InputPState*. Then to include *InputPState* it shall also link *pclosed* and *popen*. Since both constants are from the *PState* definition, *PState* should be linked to CSP as well.

Therefore, after we get the cross-reference map, our translation begins at the behavioural part from channel declarations and channel set declarations to the main actions of basic processes and other processes declarations. Finally, these behavioural constructs along with other definitions used in them (according to the map) will be translated to the final CSP by Φ Rule 1 to 38.

6.4 Ω Module

According to Figure 6.1, after the rewrite stage, the rewritten *Circus* AST_{+R} is translated to the final Z by Ω_1 and Ω_2.

Step 1, merge state paragraphs from all basic processes into one final state paragraph *State*, identify all initial schemas and merge them into one final initial paragraph *Init*, and merge other schemas as well by Ω_1 Rule 1.

Step 2, these new state paragraph, initial paragraph, and other schemas form a new ISO Standard Z model which is an AST in ISO Z format, ASTZ.

Step 3, then use the Z Pretty Printer in CZT to output this model in the LaTeX markup.

Step 4, after that, the Z LaTeX markup is fed into the Z parser and typechecker in CZT to check whether the model is free of syntax and type errors. If errors are detected, then we need to check the input *Circus* model and the translation rules to find out the causes.

Step 5, if no error is found, then a $ASTZ_+$ in Z is output from the Z typechecker.

Step 6, finally the Ω_2 module translates the inputs—$ASTZ_+$ and the Z LaTeX markup—to the final Z in ZRM LaTeX markup by Ω_2 Rule 1 to 11.

6.5 Other Considerations

6.5.1 Configuration File

A configuration file, named `config.properties`, is introduced in our translator to keep configuration constants, the main process, the path to CSP libraries, and other axiomatic definitions in the model. These information facilitates the translator to output the right Z and CSP models finally. One example of a configuration file is shown as follows.

```
# Configuration for Circus2ZCSP
CONF_MININT = 0
CONF_MAXINT = 70
CONF_MAXINS = 3
CONF_GIVEN_SET_INST_NO = 3
MAIN = SteamBoiler
CSPLIBSPATH =
MAX_NUM = 70
# capacity of boiler in litre
C = 70
# capacity of boiler in litre
M_1 = 7
N_1 = 14
N_2 = 56
M_2 = 63
# capacity of pump [litre/second]
P = 1
# litre/sec
W = 10
# litre/sec^2
U_1 = 1
# litre/sec^2
U_2 = 1
```

In a configuration file, the line starting with # is regarded as a comment line. This configuration file is used in the steam boiler case. It contains several basic configuration constants starting with `CONF\_`, `MAIN` for the main process, `CSPLIBSPATH` for the path to CSP libraries, and other constants only used in a specific model. The section below illustrates how these other constants are used in the translator.

6.5.2 Axiomatic Definitions

An axiomatic definition introduces a set of global variables with a constraint on them. For example, the definition below used in the steam boiler case defines four constants (M_1, N_1, N_2, and M_2) with a constraint (the previous constant is less than or equal to the next one).

$$M\_1, N\_1, N\_2, M\_2 : \mathbb{N}$$
$$M\_1 \leq N\_1 \leq N\_2 \leq M\_2$$

When the definition is linked to constants in CSP, they must be instantiated because constants in CSP are concrete. In addition, the instances in CSP must match their values in Z. Thus our approach is designed to model-check only one instance of all constants. We introduce a configuration file (Figure 6.3a), that stores the specific values for all constants, into our translator. At first, the axiomatic definition in *Circus* is rewritten by appending additional constraints from the configuration file into its predicate part, which results in the axiomatic definition in Figure 6.3b. Then the definition is mapped to the same axiomatics in Z by the Ω function and constants in CSP shown in Figure 6.3c by Φ Rule 3. The consistency of constants in Z and CSP is therefore preserved.

```
M_1 = 7
N_1 = 14
N_2 = 56
M_2 = 63
```

(a) Configuration File

$$M\_1, N\_1, N\_2, M\_2 : \mathbb{N}$$
$$M\_1 \leq N\_1 \leq N\_2 \leq M\_2$$
$$M\_1 = 7 \land N\_1 = 14$$
$$N\_2 = 56 \land M\_2 = 63$$

(b) Axiomatic Definition in Z

```
M_1 = 7
N_1 = 14
N_2 = 56
M_2 = 63
```

(c) Constants in CSP

Figure 6.3: Translation of Axiomatic Definition

6.6 Limitations

- For schemas as predicates in the behavioural part of *Circus*, it has been translated only in two constructs: predicate as boolean expressions in channel output expressions and predicate as conditions in guarded actions.
- Operator templates are not supported.
- Iterated sequential composition of actions is not supported because T in its declaration $x : T$ is regarded as a set in CZT and actually it should be a sequence.
- For axiomatic definitions, if they are used in CSP, they have to be instantiated to become concrete and specific in advance. Our solution cannot cope with loose constants.

6.7 Summary

A translator, *Circus2ZCSP*, is developed to translate *Circus* models to $CSP \parallel_B Z$ models automatically. This translator has been used in three case studies in Section 7. In this chapter, we present its translation procedure and give a description of each stage in detail. A number of challenges we encountered in the development of the translator, along with their solutions, are illustrated as well. In the end, the limitations of this translator are also listed.

Chapter 7

Case Studies

After the link defined in Chapter 4, its soundness given in Chapter 5, and a translator presented in Chapter 6, in order to illustrate the usability of our approach, three case studies are shown in this chapter. We introduce case studies from a simple and typical *Circus* example, the reactive buffer [64], to a more complicated Electronic Shelf Edge Label (ESEL) system that is an abstraction of real systems, and finally to the steam boiler control system [65], a real industry control system. Basically, the complete *Circus* models and the resultant final $CSP \parallel_B Z$ models of all three cases are given in Appendix G, Appendix H, and Appendix I. These models are displayed without additional description, therefore readers can focus on models themselves and will not be distracted by description.

Individually, for the buffer, since its specification and implementation models are simple and easy to be understood, Section 7.1 gives more about the stepwise application of our link rules, and model checking considerations and results. While the ESEL system is new and original in this document, therefore Section 7.2 provides a description of the system in depth, and along with its model checking results. Finally, since the steam boiler control system is well known and there has been a *Circus* solution [66, 67] for the system, in Section 7.3 we mainly emphasize on explanation of how our model checking solution can help to find the problems in the original design, correct them, and describe the problems and challenges of its model checking and animation.

7.1 Bounded Reactive Buffer

The bounded buffer is a typical example used in Z and *Circus*. A development of the bounded buffer module from specification, design, implementation to the final executable code is shown in the book [63, Chapter 22]. A similar example of a bounded reactive buffer [64] is a case study to illustrate the refinement strategy of *Circus*. It has been developed from its specification [64, Figure 1] to the final distributed cached-head ring buffer [64, Section 7.5 and 7.6]. Our buffer case in this section is based on the specification and the final ring buffer of this example. Furthermore, the *map* solution [55] of model checking *Circus* by *mapping* to CSP also uses this example as a case study.

7.1.1 Buffer Specification

The specification, *BufferSpec*, in *Circus* is shown in Appendix G. The size of the buffer is bounded by *maxbuff*. Its behaviour is specified by the main action: *buff* is initialized to be empty and its size is equal to zero; after that, it provides *input* (in case the buffer is not full) and *output* (in case the buffer is not empty) events to its environment continuously. Accordingly, it buffers the input message in its end and increments its size by one, or outputs its head and decreases its size by one.

The stepwise application of our rules to this model is displayed as follows. Provided that the configuration file has the following content (see Section 7.1.3.3 for details about guidance on how to choose concrete values for constants in the configuration file).

```
# Configuration for Circus2ZCSP
CONF_MININT = 0
CONF_MAXINT = 3
CONF_MAXINS = 5
CONF_GIVEN_SET_INST_NO = 3
maxbuff = 5
MAIN = Buffer
CSPLIBSPATH =
```

According to the translation procedure in Figure 6.1, after parsing and typechecking, the first step is to rewrite it by R_{wrt} rules.

7.1.1.1 Rewrite by R_{wrt}

The rewrite procedure is shown in Figure 6.2.

- Since this specification only has standard toolkit `circus_toolkit` as its parent section, the section resolving stage just keeps it unchanged.
- And all identifiers comply with the pattern required, so this check succeeds.
- There are no global schemas, and therefore schema localisation will not change anything.
- The *maxbuff* is an axiomatic definition. According to R_{wrt} Rule 5, since it is referred to in the behavioural part, such as the definition of the action *Input*, it has to be instantiated to 5, the value from *maxbuff* in the configuration file. Finally, the axiomatic definition becomes

$$\begin{array}{|l} maxbuff : \mathbb{N}_1 \\ \hline maxbuff = 5 \end{array}$$

- The rewrite of the channel declarations for *input* and *output* keeps them unchanged according to R_{wrt} Rule 6.
- After that, since there are no schemas as predicates, the *Schemas as Predicates to Predicates* stage also has no effect on the model.
- According to R_{wrt} Rule 17, the rewrite of a process definition is defined as the rewrite of its body. Therefore, the rewrite of the *Buffer* process is equal to rewrite its body: an explicitly defined process.
- In order to apply R_{wrt} Rule 23 to add extra operation schemas to retrieve the values of state components, firstly we need to get a list of state variables and their corresponding types: *buff* and *size* with the types $\operatorname{seq} \mathbb{N}$ and $0 \mathrel{..} maxbuff$ separately. Thus two operation schemas below are added within the process.

$$Op\_buff == [\,\Xi BufferState \,;\, buff! : \operatorname{seq} \mathbb{N} \mid buff! = buff\,]$$
$$Op\_size == [\,\Xi BufferState \,;\, size! : 0 \mathrel{..} maxbuff \mid size! = size\,]$$

- Then state components, schema paragraphs, action paragraphs, and all their references are renamed by R_{wrt} Rule 24. After being renamed, the *Buffer* process is shown in Figure 7.1.

- After that, the main action is rewritten by R_{wrt} Rules in Section 4.3.17.3. The stepwise application of action rewrite rules is demonstrated in Equation 7.1 to Equation 7.3.
- Action definitions of *Buffer_Input* and *Buffer_Output* are removed according to *Rewrite Stage Four* .
- Finally, the *Buffer* process after rewrite is illustrated in Figure 7.2.

$$
\begin{array}{l}
\mathbf{process}\ Buffer \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ Buffer\_BufferState == [\, Buffer\_buff : \mathrm{seq}\,\mathbb{N}; \\
\qquad Buffer\_size : 0 \mathrel{..} maxbuff \mid Buffer\_size = \#\, Buffer\_buff \leq maxbuff \,] \\
\quad Buffer\_BufferInit == [\, (Buffer\_BufferState)' \mid \\
\qquad Buffer\_buff' = \langle\rangle \land Buffer\_size' = 0 \,] \\
\quad Buffer\_InputCmd == [\, \Delta Buffer\_BufferState \,;\, x? : \mathbb{N} \mid \\
\qquad Buffer\_size < maxbuff \land Buffer\_buff' = Buffer\_buff \frown \langle x? \rangle \land \\
\qquad Buffer\_size' = Buffer\_size + 1 \,] \\
\quad Buffer\_Input \mathrel{\widehat=} (size < maxbuff) \,\&\, input?x \rightarrow \big(Buffer\_InputCmd\big) \\
\quad Buffer\_OutputCmd == [\, \Delta Buffer\_BufferState \mid Buffer\_size > 0 \land \\
\qquad Buffer\_buff' = tail\ Buffer\_buff \land Buffer\_size' = Buffer\_size - 1 \,] \\
\quad Buffer\_Output \mathrel{\widehat=} \\
\qquad (size > 0) \,\&\, output!(head\ Buffer\_buff) \rightarrow \big(Buffer\_OutputCmd\big) \\
\quad Buffer\_Op\_buff == [\, \Xi Buffer\_BufferState \,;\, buff! : \mathrm{seq}\,\mathbb{N} \mid buff! = Buffer\_buff \,] \\
\quad Buffer\_Op\_size == [\, \Xi Buffer\_BufferState \,;\, size! : 0 \mathrel{..} maxbuff \mid \\
\qquad size! = Buffer\_size \,] \\
\quad \bullet \big(Buffer\_BufferInit\big) \,;\, (\mu X \bullet (Buffer\_Input \Box Buffer\_Output) \,;\, X) \\
\mathbf{end}
\end{array}
$$

Figure 7.1: Renamed buffer specification

The rewrite of its main action

$$
\begin{array}{l}
R_{wrt}\left(\big(Buffer\_BufferInit\big) \,;\, (\mu X \bullet (Buffer\_Input \Box Buffer\_Output) \,;\, X)\right) \\
\quad = \big(Buffer\_BufferInit\big) \,;\, R_{wrt}\,(\mu X \bullet (Buffer\_Input \Box Buffer\_Output) \,;\, X) \\
\hfill [R_{wrt}\ \text{Rule 29 and 25}] \\
\quad = \big(Buffer\_BufferInit\big) \,;\, (\mu X \bullet R_{wrt}\,((Buffer\_Input \Box Buffer\_Output) \,;\, X)) \\
\hfill [R_{wrt}\ \text{Rule 34}] \\
\quad = \big(Buffer\_BufferInit\big) \,;\, \mu X \bullet \left(\begin{array}{l} R_{pre}\,(Buffer\_Input \Box Buffer\_Output) \rightarrow \\ \left(\begin{array}{l} R_{post}\,(Buffer\_Input \Box Buffer\_Output) \\ ;R_{wrt}\,(X) \end{array} \right) \end{array} \right) \\
\hfill [R_{wrt}\ \text{Rule 29}] \\
\quad = \big(Buffer\_BufferInit\big) \,;\, \mu X \bullet \left(\begin{array}{l} R_{pre}\,(Buffer\_Input \Box Buffer\_Output) \rightarrow \\ (R_{post}\,(Buffer\_Input \Box Buffer\_Output) \,;\, X) \end{array} \right) \\
\hfill [R_{wrt}\ \text{Rule 34}]
\end{array}
$$

$$= \left(\begin{array}{l} (Buffer\_BufferInit) ; \mu X \bullet \\ \left(\begin{array}{l} (Buffer\_Op\_size) \rightarrow (Buffer\_Op\_buff) \rightarrow \\ \left(\left(\begin{array}{l} (size < maxbuff) \mathbin{\&} input?x \rightarrow (Buffer\_InputCmd) \\ \Box \\ (size > 0) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow \\ \quad (Buffer\_OutputCmd) \end{array} \right) ; X \right) \end{array} \right) \end{array} \right)$$

[Equation 7.1]

The rewrite of the external choice in the main action

$$R_{wrt}\,(Buffer\_Input \;\Box\; Buffer\_Output)$$

$$= \left(\begin{array}{l} R_{mrg}\,(R_{pre}\,(Buffer\_Input), R_{pre}\,(Buffer\_Output)) \rightarrow \\ (R_{post}\,(Buffer\_Input) \;\Box\; R_{post}\,(Buffer\_Output)) \end{array} \right) \qquad [R_{wrt} \text{ Rule 30}]$$

$$= \left(\begin{array}{l} R_{mrg}\left((Buffer\_Op\_size), (Buffer\_Op\_size) \rightarrow (Buffer\_Op\_buff)\right) \rightarrow \\ \left(\begin{array}{l} \left((size < maxbuff) \mathbin{\&} input?x \rightarrow (Buffer\_InputCmd)\right) \\ \Box \\ \left((size > 0) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow (Buffer\_OutputCmd)\right) \end{array} \right) \end{array} \right)$$

[Equation 7.2 and 7.3]

$$= \left(\begin{array}{l} (Buffer\_Op\_size) \rightarrow (Buffer\_Op\_buff) \rightarrow \\ \left(\begin{array}{l} \left((size < maxbuff) \mathbin{\&} input?x \rightarrow (Buffer\_InputCmd)\right) \\ \Box \\ \left((size > 0) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow (Buffer\_OutputCmd)\right) \end{array} \right) \end{array} \right)$$

[Definition 4.3.2]

(7.1)

The rewrite of the action *Buffer_Input*

$$R_{wrt}\,(Buffer\_Input)$$

$$= R_{wrt}\left((size < maxbuff) \mathbin{\&} input?x \rightarrow (Buffer\_InputCmd)\right) \qquad [R_{wrt} \text{ Rule 35}]$$

$$= \left(\begin{array}{l} R_{mrg}\left(R_{pre}\,(size < maxbuff), R_{pre}\left(input?x \rightarrow (Buffer\_InputCmd)\right)\right) \rightarrow \\ \left((size < maxbuff) \mathbin{\&} R_{post}\left(input?x \rightarrow (Buffer\_InputCmd)\right)\right) \end{array} \right)$$

[R_{wrt} Rule 28]

$$= \left(\begin{array}{l} R_{pre}\,(size < maxbuff) \rightarrow \\ \left((size < maxbuff) \mathbin{\&} input?x \rightarrow R_{wrt}\left((Buffer\_InputCmd)\right)\right) \end{array} \right)$$

[R_{wrt} Rule 27 and Definition 4.3.2]

$$= \left(\begin{array}{l} (Buffer\_Op\_size) \rightarrow \\ \left((size < maxbuff) \mathbin{\&} input?x \rightarrow (Buffer\_InputCmd)\right) \end{array} \right)$$

[Definition 4.3.1 and R_{wrt} Rule 25]

(7.2)

The rewrite of the action *Buffer_Output*

$$R_{wrt}\,(Buffer\_Output)$$

$$
\begin{aligned}
&= R_{wrt}\left(\left(\mathit{size} > 0\right) \,\&\, \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \left(\mathit{Buffer\_OutputCmd}\right)\right) \\
&\hfill [R_{wrt} \text{ Rule 35}] \\
&= \left(\begin{array}{l}
R_{mrg}\left(R_{pre}\left(\mathit{size} > 0\right), R_{pre}\left(\begin{array}{l}\mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \\ \left(\mathit{Buffer\_OutputCmd}\right)\end{array}\right)\right) \rightarrow \\
\left(\left(\mathit{size} > 0\right) \,\&\, R_{post}\left(\mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \left(\mathit{Buffer\_OutputCmd}\right)\right)\right)
\end{array}\right) \\
&\hfill [R_{wrt} \text{ Rule 28}] \\
&= \left(\begin{array}{l}
\left(\mathit{Buffer\_Op\_size}\right) \rightarrow \left(\mathit{Buffer\_Op\_buff}\right) \rightarrow \\
\left(\left(\mathit{size} > 0\right) \,\&\, \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow R_{wrt}\left(\left(\mathit{Buffer\_OutputCmd}\right)\right)\right)
\end{array}\right) \\
&\hfill [R_{wrt} \text{ Rule 27, Definition 4.3.1, and Definition 4.3.2}] \\
&= \left(\begin{array}{l}
\left(\mathit{Buffer\_Op\_size}\right) \rightarrow \left(\mathit{Buffer\_Op\_buff}\right) \rightarrow \\
\left(\left(\mathit{size} > 0\right) \,\&\, \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \left(\mathit{Buffer\_OutputCmd}\right)\right)
\end{array}\right) \\
&\hfill [R_{wrt} \text{ Rule 25}]
\end{aligned}
\tag{7.3}
$$

$$
\begin{array}{l}
\textbf{process}\ \mathit{Buffer} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{Buffer\_BufferState} == [\,\mathit{Buffer\_buff} : \mathrm{seq}\ \mathbb{N}; \\
\qquad \mathit{Buffer\_size} : 0 \mathinner{.\,.} \mathit{maxbuff} \mid \mathit{Buffer\_size} = \#\, \mathit{Buffer\_buff} \leq \mathit{maxbuff}\,] \\
\quad \mathit{Buffer\_BufferInit} == [\,(\mathit{Buffer\_BufferState})' \mid \\
\qquad \mathit{Buffer\_buff}' = \langle\rangle \wedge \mathit{Buffer\_size}' = 0\,] \\
\quad \mathit{Buffer\_InputCmd} == [\,\Delta \mathit{Buffer\_BufferState}\,; x? : \mathbb{N} \mid \\
\qquad \mathit{Buffer\_size} < \mathit{maxbuff} \wedge \mathit{Buffer\_buff}' = \mathit{Buffer\_buff} \frown \langle x? \rangle \wedge \\
\qquad \mathit{Buffer\_size}' = \mathit{Buffer\_size} + 1\,] \\
\quad \mathit{Buffer\_OutputCmd} == [\,\Delta \mathit{Buffer\_BufferState} \mid \mathit{Buffer\_size} > 0 \wedge \\
\qquad \mathit{Buffer\_buff}' = \mathit{tail}\ \mathit{Buffer\_buff} \wedge \mathit{Buffer\_size}' = \mathit{Buffer\_size} - 1\,] \\
\quad \mathit{Buffer\_Op\_buff} == [\,\Xi \mathit{Buffer\_BufferState}\,; \mathit{buff}! : \mathrm{seq}\ \mathbb{N} \mid \mathit{buff}! = \mathit{Buffer\_buff}\,] \\
\quad \mathit{Buffer\_Op\_size} == [\,\Xi \mathit{Buffer\_BufferState}\,; \mathit{size}! : 0 \mathinner{.\,.} \mathit{maxbuff} \mid \\
\qquad \mathit{size}! = \mathit{Buffer\_size}\,] \\
\quad \bullet \left(\begin{array}{l}
\left(\mathit{Buffer\_BufferInit}\right); \mu X \bullet \\
\left(\begin{array}{l}
\left(\mathit{Buffer\_Op\_size}\right) \rightarrow \left(\mathit{Buffer\_Op\_buff}\right) \rightarrow \\
\left(\left(\begin{array}{l}
\left(\mathit{size} < \mathit{maxbuff}\right) \,\&\, \mathit{input}?x \rightarrow \left(\mathit{Buffer\_InputCmd}\right) \\
\Box \\
\left(\mathit{size} > 0\right) \,\&\, \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \\
\qquad \left(\mathit{Buffer\_OutputCmd}\right)
\end{array}\right); X\right)
\end{array}\right)
\end{array}\right) \\
\textbf{end}
\end{array}
$$

Figure 7.2: Buffer specification after action rewrite

After rewriting, the rewritten buffer specification is to be translated to Z and CSP separately by Ω and Φ.

7.1.1.2 The Behavioural Part

Then the behavioural part of the rewritten model in Figure 7.2 is translated by the Φ function to get a CSP specification.

The first step is to decide which constructs should be linked to CSP. Obviously, the constant *maxbuff* and the channel declaration should be translated.

The translation of

$$\begin{array}{|l} \mathit{maxbuff} : \mathbb{N}_1 \\ \hline \mathit{maxbuff} = 5 \end{array}$$

results in maxbuff = 5 according to Φ Rule 3. And the translation of the channel declaration is displayed as follows.

$$\begin{aligned} & \Phi\,(\mathbf{channel}\ \mathit{input}, \mathit{output} : \mathbb{N}) \\ & = \mathit{channelinput}, \mathit{output} : \Phi\,(\mathbb{N}) && [\Phi \text{ Rule } 4] \\ & = \mathit{channelinput}, \mathit{output} : \mathit{Nat} && [\Phi \text{ Rule 1 and Table D.4}] \end{aligned}$$

Then the process *Buffer* is linked.

$$\begin{aligned} & \Phi\,(\mathit{Buffer}) \\ & = \left(\begin{array}{l} \mathit{Buffer} = \\ \Phi \left(\begin{array}{l} \left(\mathit{Buffer\_BufferInit}\right) ; \mu X \bullet \\ \left(\begin{array}{l} \left(\mathit{Buffer\_Op\_size}\right) \rightarrow \left(\mathit{Buffer\_Op\_buff}\right) \rightarrow \\ \left(\left(\begin{array}{l} (\mathit{size} < \mathit{maxbuff}) \mathbin{\&} \mathit{input}?x \rightarrow \\ \quad \left(\mathit{Buffer\_InputCmd}\right) \\ \Box \\ (\mathit{size} > 0) \mathbin{\&} \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \\ \quad \left(\mathit{Buffer\_OutputCmd}\right) \end{array} \right) ; X \right) \end{array} \right) \end{array} \right) \end{array} \right) \\ & && [\Phi \text{ Rule } 8] \\ & = \left(\begin{array}{l} \mathit{Buffer} = \Phi\left(\left(\mathit{Buffer\_BufferInit}\right)\right) ; \\ \Phi \left(\begin{array}{l} \mu X \bullet \\ \left(\begin{array}{l} \left(\mathit{Buffer\_Op\_size}\right) \rightarrow \left(\mathit{Buffer\_Op\_buff}\right) \rightarrow \\ \left(\left(\begin{array}{l} (\mathit{size} < \mathit{maxbuff}) \mathbin{\&} \mathit{input}?x \\ \quad \rightarrow \left(\mathit{Buffer\_InputCmd}\right) \\ \Box \\ (\mathit{size} > 0) \mathbin{\&} \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \\ \quad \left(\mathit{Buffer\_OutputCmd}\right) \end{array} \right) ; X \right) \end{array} \right) \end{array} \right) \end{array} \right) \\ & && [\Phi \text{ Rule } 26] \\ & = \left(\begin{array}{l} \mathit{channel}\ \mathit{Buffer\_BufferInit} \\ \mathit{channel}\ \mathit{Buffer\_BufferInit\_fOp} \\ \mathit{HIDE\_CSPB} = \{\!|\mathit{Buffer\_BufferInit}, \mathit{Buffer\_BufferInit\_fOp}|\!\} \\ \mathit{Buffer} = (\mathit{Buffer\_BufferInit} \rightarrow \mathit{SKIP} \Box \mathit{Buffer\_BufferInit\_fOp} \rightarrow \mathbf{div}) ; \\ \Phi \left(\begin{array}{l} \mu X \bullet \\ \left(\begin{array}{l} \left(\mathit{Buffer\_Op\_size}\right) \rightarrow \left(\mathit{Buffer\_Op\_buff}\right) \rightarrow \\ \left(\left(\begin{array}{l} (\mathit{size} < \mathit{maxbuff}) \mathbin{\&} \mathit{input}?x \rightarrow \\ \quad \left(\mathit{Buffer\_InputCmd}\right) \\ \Box \\ (\mathit{size} > 0) \mathbin{\&} \mathit{output}!(\mathit{head}\ \mathit{Buffer\_buff}) \rightarrow \\ \quad \left(\mathit{Buffer\_OutputCmd}\right) \end{array} \right) ; X \right) \end{array} \right) \end{array} \right) \end{array} \right) \\ & && [\Phi \text{ Rule } 21] \end{aligned}$$

$$
= \left(\begin{array}{l}
channel\ Buffer\_BufferInit \\
channel\ Buffer\_BufferInit\_fOp \\
channel\ Buffer\_Op\_size : \{0..maxbuff\} \\
channel\ Buffer\_Op\_buff : fseq\,(Nat) \\
channel\ Buffer\_InputCmd : Nat \\
channel\ Buffer\_InputCmd\_fOp : Nat \\
channel\ Buffer\_OutputCmd \\
channel\ Buffer\_OutputCmd\_fOp \\
HIDE\_CSPB = \{\!| \left(\begin{array}{l} Buffer\_BufferInit, Buffer\_BufferInit\_fOp, \\ Buffer\_Op\_size, Buffer\_Op\_buff, \\ Buffer\_InputCmd, Buffer\_InputCmd\_fOp, \\ Buffer\_OutputCmd, Buffer\_OutputCmd\_fOp \end{array}\right) |\!\} \\
Buffer = (Buffer\_BufferInit \to SKIP \Box Buffer\_BufferInit\_fOp \to \mathbf{div}); \\
let\ X = Buffer\_Op\_size?size \to Buffer\_Op\_buff?buff \to \\
\quad \left(\left(\begin{array}{l}
\left(\begin{array}{l} (size < maxbuff)\ \& \\ \left(input?x \to \left(\begin{array}{l} Buffer\_InputCmd!x \to SKIP \\ \Box \\ Buffer\_InputCmd\_fOp!x \to \mathbf{div} \end{array} \right) \right) \end{array} \right) \\
\Box \\
\left(\begin{array}{l} (size > 0)\ \&\ \ output!(head(Buffer\_buff)) \to \\ \left(\begin{array}{l} Buffer\_OutputCmd \to SKIP \\ \Box \\ Buffer\_OuputCmd\_fOp \to \mathbf{div} \end{array} \right) \end{array} \right)
\end{array} \right) ; X \right) \\
within\ X
\end{array}\right)
$$

[Equation 7.5]

(7.4)

And another schema *Buffer_BufferInit_fOp*

$$Buffer\_BufferInit\_fOp = [\Xi Buffer\_BufferState \mid \neg\, \mathbf{pre}\, Buffer\_BufferInit]$$

is added within the process *Buffer* according to Φ Rule 21. Now we continue to apply rules to the recursion.

$$
\Phi \left(\begin{array}{l}
\mu X \bullet \\
\left(\begin{array}{l}
\big(Buffer\_Op\_size\big) \to \big(Buffer\_Op\_buff\big) \to \\
\left(\begin{array}{l}
(size < maxbuff)\ \&\ input?x \to \big(Buffer\_InputCmd\big) \\
\Box \\
(size > 0)\ \&\ output!(head\ Buffer\_buff) \to \\
\qquad \big(Buffer\_OutputCmd\big)
\end{array} \right) ; X
\end{array} \right)
\end{array} \right)
$$

$$
= \left(\begin{array}{l}
let\ X = \\
\quad \Phi \left(\begin{array}{l}
\big(Buffer\_Op\_size\big) \to \big(Buffer\_Op\_buff\big) \to \\
\left(\left(\begin{array}{l}
(size < maxbuff)\ \&\ input?x \to \\
\qquad \big(Buffer\_InputCmd\big) \\
\Box \\
(size > 0)\ \&\ output!(head\ Buffer\_buff) \to \\
\qquad \big(Buffer\_OutputCmd\big)
\end{array} \right) ; X \right)
\end{array} \right) \\
within\ X
\end{array} \right)
$$

[Φ Rule 34]

$$= \left(\begin{array}{l} let\ X = \\ \Phi\left(\left(Buffer\_Op\_size\right)\right) \rightarrow \Phi\left(\left(Buffer\_Op\_buff\right)\right) \rightarrow \\ \quad \Phi\left(\left(\begin{array}{l} \left(size < maxbuff\right) \mathbin{\&} input?x \rightarrow \\ \qquad \left(Buffer\_InputCmd\right) \\ \Box \\ \left(size > 0\right) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow \\ \qquad \left(Buffer\_OutputCmd\right) \end{array}\right) ; X\right) \\ within\ X \end{array}\right) \qquad [\Phi \text{ Rule } 24]$$

$$= \left(\begin{array}{l} channel\ Buffer\_Op\_size : \Phi\left(0\,..\,maxbuff\right) \\ channel\ Buffer\_Op\_buff\ : \Phi\left(\mathrm{seq}\ \mathbb{N}\right) \\ HIDE\_CSPB = \{\!|\cdots, Buffer\_Op\_size, Buffer\_Op\_buff\,|\!\} \\ let\ X = \\ Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \quad \Phi\left(\left(\begin{array}{l} \left(size < maxbuff\right) \mathbin{\&} input?x \rightarrow \\ \qquad \left(Buffer\_InputCmd\right) \\ \Box \\ \left(size > 0\right) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow \\ \qquad \left(Buffer\_OutputCmd\right) \end{array}\right) ; X\right) \\ within\ X \end{array}\right) \qquad [\Phi \text{ Rule } 22]$$

$$= \left(\begin{array}{l} channel\ Buffer\_Op\_size : \{0..maxbuff\} \\ channel\ Buffer\_Op\_buff\ : fseq\,(Nat) \\ HIDE\_CSPB = \{\!|\cdots, Buffer\_Op\_size, Buffer\_Op\_buff\,|\!\} \\ let\ X = \\ Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \quad \left(\Phi\left(\begin{array}{l} \left(size < maxbuff\right) \mathbin{\&} input?x \rightarrow \\ \qquad \left(Buffer\_InputCmd\right) \\ \Box \\ \left(size > 0\right) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow \\ \qquad \left(Buffer\_OutputCmd\right) \end{array}\right) ; \Phi\,(X)\right) \\ within\ X \end{array}\right)$$

[Φ Rule 1, Table D.5, Table D.8, and Φ Rule 26]

$$= \left(\begin{array}{l} channel\ Buffer\_Op\_size : \{0..maxbuff\} \\ channel\ Buffer\_Op\_buff\ : fseq\,(Nat) \\ HIDE\_CSPB = \{\!|\cdots, Buffer\_Op\_size, Buffer\_Op\_buff\,|\!\} \\ let\ X = \\ Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \quad \left(\left(\begin{array}{l} \Phi\left(\begin{array}{l} \left(size < maxbuff\right) \mathbin{\&} input?x \rightarrow \\ \qquad \left(Buffer\_InputCmd\right) \end{array}\right) \\ \Box \\ \Phi\left(\begin{array}{l} \left(size > 0\right) \mathbin{\&} output!(head\ Buffer\_buff) \rightarrow \\ \qquad \left(Buffer\_OutputCmd\right) \end{array}\right) \end{array}\right) ; X\right) \\ within\ X \end{array}\right)$$

[Φ Rule 27 and 34]

$$= \left(\begin{array}{l} channel\ Buffer\_Op\_size : \{0..maxbuff\} \\ channel\ Buffer\_Op\_buff : fseq\,(Nat) \\ HIDE\_CSPB = \{| \cdots , Buffer\_Op\_size, Buffer\_Op\_buff |\} \\ let\ X = \\ Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \left(\left(\begin{array}{l} \left(\begin{array}{l} \Phi\,(size < maxbuff)\ \ \& \\ \Phi\left(input?x \rightarrow \left(Buffer\_InputCmd\right)\right) \end{array} \right) \\ \Box \\ \left(\begin{array}{l} \Phi\,(size > 0)\ \ \& \\ \Phi \left(\begin{array}{l} output!(head\ Buffer\_buff) \rightarrow \\ \left(Buffer\_OutputCmd\right) \end{array} \right) \end{array} \right) \end{array} \right) ; X \right) \\ within\ X \end{array} \right) \qquad [\Phi\ \text{Rule 25}]$$

$$= \left(\begin{array}{l} channel\ Buffer\_Op\_size : \{0..maxbuff\} \\ channel\ Buffer\_Op\_buff : fseq\,(Nat) \\ HIDE\_CSPB = \{| \cdots , Buffer\_Op\_size, Buffer\_Op\_buff |\} \\ let\ X = \\ Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \left(\left(\begin{array}{l} \left(\begin{array}{l} (size < maxbuff)\ \ \& \\ \left(input?x \rightarrow \Phi\left(\left(Buffer\_InputCmd\right)\right)\right) \end{array} \right) \\ \Box \\ \left(\begin{array}{l} (size > 0)\ \ \& \\ \left(\begin{array}{l} output!(head(Buffer\_buff)) \rightarrow \\ \Phi\left(\left(Buffer\_OutputCmd\right)\right) \end{array} \right) \end{array} \right) \end{array} \right) ; X \right) \\ within\ X \end{array} \right)$$

[Φ Rule 24, Table D.4, and Table D.8]

$$= \left(\begin{array}{l} channel\ Buffer\_Op\_size : \{0..maxbuff\} \\ channel\ Buffer\_Op\_buff : fseq\,(Nat) \\ channel\ Buffer\_InputCmd : Nat \\ channel\ Buffer\_InputCmd\_fOp : Nat \\ channel\ Buffer\_OutputCmd \\ channel\ Buffer\_OutputCmd\_fOp \\ HIDE\_CSPB = \{| \left(\begin{array}{l} \cdots , Buffer\_Op\_size, Buffer\_Op\_buff, \\ Buffer\_InputCmd, Buffer\_InputCmd\_fOp, \\ Buffer\_OutputCmd, Buffer\_OutputCmd\_fOp \end{array} \right) |\} \\ let\ X = \\ Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \left(\left(\begin{array}{l} \left(\begin{array}{l} (size < maxbuff)\ \ \& \\ \left(input?x \rightarrow \left(\begin{array}{l} Buffer\_InputCmd!x \rightarrow SKIP \\ \Box \\ Buffer\_InputCmd\_fOp!x \rightarrow \mathbf{div} \end{array} \right) \right) \end{array} \right) \\ \Box \\ \left(\begin{array}{l} (size > 0)\ \ \&\ \ output!(head(Buffer\_buff)) \rightarrow \\ \left(\begin{array}{l} Buffer\_OutputCmd \rightarrow SKIP \\ \Box \\ Buffer\_OuputCmd\_fOp \rightarrow \mathbf{div} \end{array} \right) \end{array} \right) \end{array} \right) ; X \right) \\ within\ X \end{array} \right)$$

[Φ Rule 21 and Table D.4]

(7.5)

There are other schemas added within the *Buffer* process according to Φ Rule 21 and Φ

Rule 22.

$$\begin{array}{l}
Buffer\_InputCmd\_fOp = [\Xi Buffer\_BufferState \,;\, x? : \mathbb{N} \mid \neg\, \mathbf{pre}\, Buffer\_InputCmd] \\
Buffer\_OutputCmd\_fOp = [\Xi Buffer\_BufferState \mid \neg\, \mathbf{pre}\, Buffer\_OutputCmd]
\end{array}$$

Finally, we get the CSP specification: the *maxbuff* constant, the channel declaration, and the Equation 7.4. The complete CSP model is shown in Appendix G.1.2. In addition, three additional schemas are within the process *Buffer*, and the updated *Buffer* process is displayed in Figure 7.3. In the figure, since the main action has been translated, it is omitted.

$$\begin{array}{l}
\mathbf{process}\ Buffer \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ Buffer\_BufferState == [\, Buffer\_buff : \operatorname{seq} \mathbb{N}; \\
\qquad Buffer\_size : 0 \mathinner{..} maxbuff \mid Buffer\_size = \# Buffer\_buff \leq maxbuff \,] \\
\quad Buffer\_BufferInit == [\, (Buffer\_BufferState)' \mid \\
\qquad Buffer\_buff' = \langle\rangle \land Buffer\_size' = 0 \,] \\
\quad Buffer\_InputCmd == [\, \Delta Buffer\_BufferState \,;\, x? : \mathbb{N} \mid \\
\qquad Buffer\_size < maxbuff \land Buffer\_buff' = Buffer\_buff \frown \langle x? \rangle \land \\
\qquad Buffer\_size' = Buffer\_size + 1 \,] \\
\quad Buffer\_OutputCmd == [\, \Delta Buffer\_BufferState \mid Buffer\_size > 0 \land \\
\qquad Buffer\_buff' = tail\ Buffer\_buff \land Buffer\_size' = Buffer\_size - 1 \,] \\
\quad Buffer\_Op\_buff == [\, \Xi Buffer\_BufferState \,;\, buff! : \operatorname{seq} \mathbb{N} \mid buff! = Buffer\_buff \,] \\
\quad Buffer\_Op\_size == [\, \Xi Buffer\_BufferState \,;\, size! : 0 \mathinner{..} maxbuff \mid \\
\qquad size! = Buffer\_size \,] \\
\quad Buffer\_BufferInit\_fOp = [\Xi Buffer\_BufferState \mid \neg\, \mathbf{pre}\, Buffer\_BufferInit] \\
\quad Buffer\_InputCmd\_fOp = [\Xi Buffer\_BufferState \,;\, x? : \mathbb{N} \mid \\
\qquad \neg\, \mathbf{pre}\, Buffer\_InputCmd] \\
\quad Buffer\_OutputCmd\_fOp = [\Xi Buffer\_BufferState \mid \neg\, \mathbf{pre}\, Buffer\_OutputCmd] \\
\quad \bullet \ldots \\
\mathbf{end}
\end{array}$$

Figure 7.3: Buffer specification after translation of behaviour

7.1.1.3 The State Part

After translation of the behavioural part, the state part is translated to Z by Ω. The procedure is given in Section 6.4. The first step is to merge states and schemas by Ω_1 Rule 1. The specification only has one process *Buffer*, so the merge is straightforward. Additionally, the initialisation schema, *Buffer_BufferInit*, is identified according to our Definition 4.4.1. Then we get the translated Z specification in ISO Standard Z as shown in Figure 7.4. It is parsed and typechecked, and finally translated to Z in ZRM by Ω_2 Rule 1 to 11. The final Z model is displayed in Appendix G.1.2.

7.1.2 Distributed Reactive Buffer

The distributed cached-head ring buffer [64], an implementation of the buffer specification, is a result of the refinement strategy development in that paper. Its complete *Circus* model is shown in Appendix G.2.1. It is composed of the process *Controller*, the process *RingCell*, the indexed ring cell process *IRCell*, and the process *Ring*. Finally the process *Buffer* is a parallel composition of the process *Ring* and the process *Controller*. In addition, there are two constants *maxbuff* and *maxring*, an abbreviation *RingIndex*, and a number of channel declarations.

$\textbf{section}\ \mathit{BufferSpec}\ \textbf{parents}\ \mathit{standard\_toolkit}$

$$\begin{array}{|l} \mathit{maxbuff} : \mathbb{N}_1 \\ \hline \mathit{maxbuff} = 5 \end{array}$$

$$\begin{array}{l}
\mathit{Buffer\_BufferState} == [\mathit{Buffer\_buff} : \mathrm{seq}\,\mathbb{N}\,;\, \mathit{Buffer\_size} : 0 \mathrel{..} \mathit{maxbuff} \mid \\
\qquad \mathit{Buffer\_size} = \#\,\mathit{Buffer\_buff} \leq \mathit{maxbuff}] \\
\mathit{State} == \mathit{Buffer\_BufferState} \\
\mathit{Init} == [\mathit{State}' \mid \mathit{Buffer\_buff}' = \langle\rangle \land \mathit{Buffer\_size}' = 0] \\
\mathit{Buffer\_BufferInit} == [(\mathit{Buffer\_BufferState})' \mid \mathit{Buffer\_buff}' = \langle\rangle \land \mathit{Buffer\_size}' = 0] \\
\mathit{Buffer\_InputCmd} == [\Delta \mathit{Buffer\_BufferState}\,;\, x? : \mathbb{N} \mid \mathit{Buffer\_size} < \mathit{maxbuff} \land \\
\qquad \mathit{Buffer\_buff}' = \mathit{Buffer\_buff} \mathbin{^\frown} \langle x? \rangle \land \mathit{Buffer\_size}' = \mathit{Buffer\_size} + 1] \\
\mathit{Buffer\_OutputCmd} == [\Delta \mathit{Buffer\_BufferState} \mid \mathit{Buffer\_size} > 0 \land \\
\qquad \mathit{Buffer\_buff}' = \mathit{tail}\ \mathit{Buffer\_buff} \land \mathit{Buffer\_size}' = \mathit{Buffer\_size} - 1] \\
\mathit{Buffer\_OP\_buff} == [\Xi \mathit{Buffer\_BufferState}\,;\, \mathit{Buffer\_buff}! : \mathrm{seq}\,\mathbb{N} \mid \\
\qquad \mathit{Buffer\_buff}! = \mathit{Buffer\_buff}] \\
\mathit{Buffer\_OP\_size} == [\Xi \mathit{Buffer\_BufferState}\,;\, \mathit{Buffer\_size}! : 0 \mathrel{..} \mathit{maxbuff} \mid \\
\qquad \mathit{Buffer\_size}! = \mathit{Buffer\_size}] \\
\mathit{Buffer\_BufferInit\_fOp} == [\Xi \mathit{Buffer\_BufferState} \mid \lnot\, \textbf{pre}\ \mathit{Buffer\_BufferInit}] \\
\mathit{Buffer\_InputCmd\_fOp} == [\Xi \mathit{Buffer\_BufferState}\,;\, x? : \mathbb{N} \mid \lnot\, \textbf{pre}\ \mathit{Buffer\_InputCmd}] \\
\mathit{Buffer\_OutputCmd\_fOp} == [\Xi \mathit{Buffer\_BufferState} \mid \lnot\, \textbf{pre}\ \mathit{Buffer\_OutputCmd}]
\end{array}$$

Figure 7.4: Buffer Z specification in ISO Standard Z

This section will not give all details of the stepwise application of our rules to this model like that to the buffer specification, but more focus on the differences between both models.

Provided that the configuration file has the following content.

```
# Configuration for Circus2ZCSP
CONF_MININT = 0
CONF_MAXINT = 3
CONF_MAXINS = 5
CONF_GIVEN_SET_INST_NO = 3
maxring = 4
maxbuff = 5
MAIN = Buffer
CSPLIBSPATH =
```

According to the translation procedure in Figure 6.1, after parsing and typechecking, the first step is to rewrite it by R_{wrt} rules.

7.1.2.1 Rewriting by R_{wrt}

The rewrite procedure is shown in Figure 6.2.

- Since this specification only has standard toolkit `circus_toolkit` as its parent section, the section resolving stage just keeps it unchanged.
- And all identifiers comply with the pattern required, so this check succeeds.

- There are no global schemas, and therefore schema localisation will not change anything.

- Two constants *maxbuff* and *maxring* are given in the axiomatic definition. According to R_{wrt} Rule 5, since they are referred to in the behavioural part, such as the definitions of the action *InputController* and the process *IRCell*, they have to be instantiated to 5 and 4, the values from *maxbuff* and *maxring* in the configuration file. Finally, the axiomatic definition becomes

$$\begin{array}{|l}
maxbuff : \mathbb{N}_1 \\
maxring : \mathbb{N}_1 \\
\hline
maxring = maxbuff - 1 \\
maxbuff = 5 \\
maxring = 4
\end{array}$$

- The rewrite of the channel declarations keeps them unchanged according to R_{wrt} Rule 6.

- Since the process *IRCell* is a renamed and indexed process, according to R_{wrt} Rule 21, it should be rewritten by the rule. The application of this rule is illustrated in Equation 7.6, where *RingIndex* is expanded to $1 \mathrel{..} maxring = \{1, 2, 3, 4\}$.

- And the indexed process invocation in the process *Ring* is rewritten by R_{wrt} Rule 15. This is illustrated in Equation 7.7.

- For other processes except basic processes, the rewrite will leave them unchanged.

- After that, since there are no schemas as predicates, the *Schemas as Predicates to Predicates* stage also has no effect on the model.

- Then the rewrite of basic processes is to rewrite its body according to R_{wrt} Rule 17. It is very similar to the write of the process *Buffer* in the buffer specification. Therefore, for brevity, it is omitted.

- Finally, the model has been rewritten.

$$\begin{array}{l}
R_{wrt}\left(\begin{array}{l}
\mathbf{process}\ IRCell \mathrel{\widehat=} \\
(\, i : RingIndex \odot RingCell\,) \\
\quad [rd\_i, wrt\_i := read, write]
\end{array}\right) \\
= \left(\begin{array}{l}
\left(\begin{array}{l}
\mathbf{process}\ \ IRCell\_1 \mathrel{\widehat=} \\
\qquad R_{wrt}\left(F_{Ren}\left(B\left(RingCell\right), \{(rd, read.1), (wrt, write.1)\}\right)\right)
\end{array}\right) \\
\ldots \\
\left(\begin{array}{l}
\mathbf{process}\ \ IRCell\_4 \mathrel{\widehat=} \\
\qquad R_{wrt}\left(F_{Ren}\left(B\left(RingCell\right), \{(rd, read.4), (wrt, write.4)\}\right)\right)
\end{array}\right)
\end{array}\right) \\
\hfill [R_{wrt}\ \text{Rule 21}]
\end{array}$$

$$
= \left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{process}\ \mathit{IRCell\_1} \mathrel{\widehat=} \\
R_{wrt} \left(\begin{array}{l}
\textbf{begin} \\
\quad \textbf{state}\ \mathit{CellState} == [\, v : \mathbb{N} \mid \mathit{true} \,] \\
\quad \mathit{CellWrite}\ ==\ [\, \Delta \mathit{CellState}\, ;\, x? : \mathbb{N} \mid v' = x? \,] \\
\quad \mathit{Read} \mathrel{\widehat=} \mathit{read}.1!v \rightarrow \textbf{Skip} \\
\quad \mathit{Write} \mathrel{\widehat=} \mathit{write}.1?x \rightarrow \big(\mathit{CellWrite}\big) \\
\quad \bullet\ (\mu X \bullet (\mathit{Read} \mathrel{\Box} \mathit{Write})\ ;\ X) \\
\textbf{end}
\end{array} \right)
\end{array} \right) \\
\ldots \\
\left(\begin{array}{l}
\textbf{process}\ \mathit{IRCell\_4} \mathrel{\widehat=} \\
R_{wrt} \left(\begin{array}{l}
\textbf{begin} \\
\quad \textbf{state}\ \mathit{CellState} == [\, v : \mathbb{N} \mid \mathit{true} \,] \\
\quad \mathit{CellWrite}\ ==\ [\, \Delta \mathit{CellState}\, ;\, x? : \mathbb{N} \mid v' = x? \,] \\
\quad \mathit{Read} \mathrel{\widehat=} \mathit{read}.4!v \rightarrow \textbf{Skip} \\
\quad \mathit{Write} \mathrel{\widehat=} \mathit{write}.4?x \rightarrow \big(\mathit{CellWrite}\big) \\
\quad \bullet\ (\mu X \bullet (\mathit{Read} \mathrel{\Box} \mathit{Write})\ ;\ X) \\
\textbf{end}
\end{array} \right)
\end{array} \right)
\end{array} \right)
$$

$$[B \text{ Definition B.2.15 and } F_{Ren} \text{ Definition B.2.14}] \tag{7.6}$$

$$
\begin{array}{ll}
R_{wrt}\left(\textbf{process}\ \mathit{Ring} \mathrel{\widehat=} \left(\mathop{|||} i : \mathit{RingIndex} \bullet \mathit{IRCell}\lfloor i \rfloor \right)\right) & \\
= \textbf{process}\ \mathit{Ring} \mathrel{\widehat=} R_{wrt}\left(\mathop{|||} i : \mathit{RingIndex} \bullet \mathit{IRCell}\lfloor i \rfloor \right) & [R_{wrt} \text{ Rule 17}] \\
= \textbf{process}\ \mathit{Ring} \mathrel{\widehat=} \left(\mathop{|||} i : \mathit{RingIndex} \bullet R_{wrt}\left(\mathit{IRCell}\lfloor i \rfloor\right) \right) & [R_{wrt} \text{ Rule 22}] \\
= \textbf{process}\ \mathit{Ring} \mathrel{\widehat=} \left(\mathop{|||} i : \mathit{RingIndex} \bullet \left(\begin{array}{ll}
 & (i = 1)\ \&\ \mathit{IRCell\_1} \\
\Box & (i = 2)\ \&\mathit{IRCell\_2} \\
\Box & (i = 3)\ \&\mathit{IRCell\_3} \\
\Box & (i = 4)\ \&\mathit{IRCell\_4}
\end{array} \right) \right) & \\
 & [R_{wrt} \text{ Rule 15}]
\end{array} \tag{7.7}
$$

7.1.2.2 The Behavioural Part

Then the behavioural part of the rewritten model is translated by the Φ function to get a CSP specification. The processes including *Controller*, *RingCell*, *IRCell_1*, ..., *IRCell_4*, *Ring*, and *Buffer* should be translated. For basic processes, it is equal to translate their main actions. And for *Ring* and *Buffer*, Φ Rule 20 and 12 are used. The final CSP model is shown in Appendix G.2.2.

7.1.2.3 The State Part

After the translation of the behavioural part, the state part is translated to Z by Ω. The procedure is given in Section 6.4.

The first step is to merge states and schemas by Ω_1 Rule 1. Since the model has a number of processes, the merge is not straightforward. The state after merge is shown in Figure 7.5, where the final state space consists of state components from all basic processes.

Additionally, the initialisation schemas are identified according to our Definition 4.4.1. For the *Controller* process, *Controller_ControllerInit* is its initialisation schema. And for other basic processes *RingCell*, *IRCell_1*, ..., and *IRCell_4*, they do not have initialisation schemas and are regarded as having the predicate *true* for their initial states. Finally, the

$$\begin{aligned}
&\mathit{Controller\_ControllerState} == [\mathit{Controller\_size} : 0 \mathrel{..} \mathit{maxbuff};\\
&\quad \mathit{Controller\_ringsize} : 0 \mathrel{..} \mathit{maxring} ; \mathit{Controller\_cache} : \mathbb{N};\\
&\quad \mathit{Controller\_top}, \mathit{Controller\_bot} : \mathit{RingIndex} \mid\\
&\quad \mathit{Controller\_ringsize} \bmod \mathit{maxring} =\\
&\qquad (\mathit{Controller\_top} - \mathit{Controller\_bot}) \bmod \mathit{maxring} \land\\
&\quad \mathit{Controller\_ringsize} = \mathit{max}\{0, \mathit{Controller\_size} - 1\}]\\
&\mathit{RingCell\_CellState} == [\mathit{RingCell\_v} : \mathbb{N} \mid \mathit{true}]\\
&\mathit{RCell\_1\_CellState} == [\mathit{IRCell\_1\_v} : \mathbb{N} \mid \mathit{true}]\\
&\mathit{RCell\_2\_CellState} == [\mathit{IRCell\_2\_v} : \mathbb{N} \mid \mathit{true}]\\
&\mathit{RCell\_3\_CellState} == [\mathit{IRCell\_3\_v} : \mathbb{N} \mid \mathit{true}]\\
&\mathit{RCell\_4\_CellState} == [\mathit{IRCell\_4\_v} : \mathbb{N} \mid \mathit{true}]\\
&\mathit{State} == \mathit{Controller\_ControllerState} \land \mathit{RingCell\_CellState} \land\\
&\quad \mathit{IRCell\_1\_CellState} \land \mathit{IRCell\_2\_CellState} \land\\
&\quad \mathit{IRCell\_3\_CellState} \land \mathit{IRCell\_4\_CellState}
\end{aligned}$$

Figure 7.5: State schema after merge

initial schema is a conjunction of predicates from all basic processes' initialisation schemas, which is shown as follows.

$$\begin{aligned}
&\mathit{Init} == [\mathit{State}' \mid \mathit{Controller\_top}' = 1 \land\\
&\quad \mathit{Controller\_bot}' = 1 \land \mathit{Controller\_size}' = 0 \land \mathit{true}]
\end{aligned}$$

For other operation schemas, such as *Controller_CacheInput*, their declaration part includes Ξ of other basic processes' state schemas to make sure it will not change the state components from other processes, and their predicates are the same as the original predicates before merge. The *Controller_CacheInput* after merge is displayed below.

$$\begin{aligned}
&\mathit{Controller\_CacheInput} == [\Delta \mathit{Controller\_ControllerState} ; x? : \mathbb{N};\\
&\quad \Xi \mathit{RingCell\_CellState} ; \Xi \mathit{IRCell\_1\_CellState};\\
&\quad \Xi \mathit{IRCell\_2\_CellState} ; \Xi \mathit{IRCell\_3\_CellState};\\
&\quad \Xi \mathit{IRCell\_4\_CellState} \mid \mathit{Controller\_size} = 0 \land\\
&\quad \mathit{Controller\_size}' = 1 \land \mathit{Controller\_cache}' = x? \land\\
&\quad \mathit{Controller\_bot}' = \mathit{Controller\_bot} \land \mathit{Controller\_top}' = \mathit{Controller\_top}]
\end{aligned}$$

After merge, we get the translated Z specification in ISO Standard Z. Then it is parsed and typechecked, and finally translated to Z in ZRM by Ω_2 Rule 1 to 11. The final Z model is displayed in Appendix G.2.2.

7.1.3 Model Checking Results

Now we have got the final $\mathit{CSP} \parallel_B Z$ models for both the buffer specification and the implementation. Both of them can be model-checked by ProB. But before performing model checking, the value of the constants *MAXINT*, *MAXINS* and *maxbuff* should be considered at first.

7.1.3.1 Maximum Instances *MAXINS* and Maximum Size of Buffer *maxbuff*

For the buffer specification, the type of *buff* is $\mathit{seq}\,\mathbb{N}$. When linked to $\mathit{CSP} \parallel_B Z$, we use *fseq* (Appendix D.3) that introduces the bound constant *MAXINS*. Finally the size of the set of finite sequences by *fseq* highly relies on the value of *MAXINS* as well as the data set s. The defined *fseq* computes the result relied on several intermediate functions (*squash*, *pfun*, *rel* and *cross*) which are defined in the functional language as well. The consumption of resources during resolution is still high. If the size of s is big and $MAXINT$ is large, ProB will take longer time to compute all possible finite sequences. For an instance, on the system having 2GB RAM and 2.5 GHz CPU, and running Ubuntu 12.04, it takes approximately thirty minutes for ProB to load the CSP program when the size of s is 4 ($MAXINT$ is set to 3) and $MAXINS$ is 5. However if the size of s is reduced to 2, we can increase $MAXINS$ to 9 to make ProB load the CSP program still in a shorter time. Alternatively, instead of using the functional language to resolve *fseq*, we can compute all finite sequences in advance by another language, let's say Perl, then include them explicitly into a set and replace *fseq* in CSP programs by this set. For example, if s is $\{0, 1\}$ and $MAXINT$ is 2, then we can get this set as $\{\langle\rangle, \langle 0\rangle, \langle 1\rangle, \langle 0, 0\rangle, \langle 0, 1\rangle, \langle 1, 0\rangle, \langle 1, 1\rangle\}$. By this way, it can reduce the program load time tremendously. But the loss of flexibility is a side effect because we have to compute this set in advance and externally (out of ProB).

For the buffer specification the value of *maxbuff* should be less than or equal to $MAXINS$.

7.1.3.2 Data Independence and $MAXINT$

Informally speaking, a reactive program is data independent if the input data of the program is changed, the behaviour of the program will not change and the only change is the values of the output data. The precise definition of data independence [90, Definition 4.1] is given: for all data domain D, available data sets Σ over D, and functions $f : D \to D'$, σ is a possible behaviour of a simple reactive program P for Σ if and only if $f(\sigma)$ is also a possible behaviour of P for $f(\Sigma)$. In other words, if a data domain D is changed to another D', the behaviour of P (the sequence of *in* and *out* events) is still the same except that the output value is changed to a function on D'.

But how to tell if a reactive program is data independent or not. A semantic study of data independence [91, Section 2.7] has established it as a simple syntactic property of terms. Lazić [91, Section 2.7] and Roscoe [8, Section 15.3.2] gave the criteria of data independence of a concurrent system P with respect to a data type T for a language combined from CSP and λ-calculus, and CSP$_M$ respectively. A program is data independent with respect to a data type T if it only does the following things with values from T:

- input them along its input channels,
- store them and copy them for later use,
  - Operations which only pass members of T around without look inside of them, such as basic polymorphic operations (tupling, list formation, etc.), are regarded as suitable operations to preserve data independence. Other functions which rely on the type of T or the size of T may not be used.
  - If a concrete value from T appears in the program text of P, it will not be data independent.
  - For CSP, if the indexing set of replicated constructs depends on T, then only replicated internal choice can appear in the program text.
- output them along its output channel without performing any 'interesting' computations to constrain what T might be, and
  - If two members of T are added to each other, this is regarded as an interesting computation because it puts a constraint on the type of T in which all members must be numbers.
- perform equality and inequality tests between them.

In addition, these criteria are not only valid for CSP but also for other typed languages. However, for different languages these criteria may have different syntactic restrictions. An example of using data independence criteria in another language is the compositional abstraction of systems modelled in CSP_Z [92], a combination of CSP and Z, by partitioning the Z part into a data independent part and a data dependent part, then converting both parts to pure CSP, finally using data independence and data abstraction to analyse the data independent part and the data dependent part of resultant CSP processes.

In this case study, the type of data (T) in both the buffer specification and the buffer implementation is $\mathbb{N}$. Through syntactic checking of both the linked buffer models in $CSP \parallel_B Z$, we conclude that they are data-independent with respect to the type of data $\mathbb{N}$ in the buffer because they input values of $\mathbb{N}$ along their *input* channels, store them in a sequence or a set of ring cells, and then output values in order along their *output* channels without any computations. And they do not perform any explicit and implicit equality tests over T, therefore they satisfy **NoEqT** [71, 91]. In addition, the *Buffer* process in the linked buffer specification satisfies **Norm** [71,91]. According to Theorem 17.2 [71], the threshold of the size of T such that the implementation is a refinement of the specification in terms of traces, failures and failures-divergences is 2. There is a similar conclusion in the book [71, p.397] that the threshold of an N-bounded buffer for any N is 2. So for the refinement check, we can set $MAXINT$ to 1 as there are two elements $\{0, 1\}$ and set $maxbuff$ to 3. Actually, we also checked the refinement when $MAXINT$ is increased to 3.

7.1.3.3 Guidance on Constants in Configuration File

An example of the configuration file for this buffer case is shown in Section 7.1.1. In the file, there are several configuration items defined. Among them, the items beginning with `CONF_` denote axiomatic definitions or constants used in the $CSP \parallel_B Z$ model.

`CONF_MININT` and `CONF_MAXINT` define the minimum integer and the maximum integer for the model checking. They make $\mathbb{Z}$ and $\mathbb{N}$ be finite. In this case, $\mathbb{N}$ is the type for elements in the buffer. Therefore, `CONF_MININT` is assigned to 0. For `CONF_MAXINT`, thanks to the previous conclusion that both the linked specification and implementation models are data independent and their thresholds of the size of $\mathbb{N}$ are 2, this constant should be at least larger than or equal to 1. In the configuration file illustrated in Section 7.1.1, we choose `CONF_MAXINT` as 3.

`CONF_MAXINS` and `maxbuff` define the maximum size of the finite sequence in a set of finite sequences given by *fseq*. The value of `CONF_MAXINS` should be larger than or equal to `maxbuff`. There is no optimal value for `CONF_MAXINS` and its choice depends on the capability of the model checker and the maximum size of the buffers to be checked. In the configuration file illustrated, both of them are set to 5, which indicates only the models for a buffer to store up to 5 elements are verified. It is possible to modify ProB to make it automatically check the models whose size is equal to $1, 2, \cdots$ and *maxbuff* individually. In other words, `maxbuff=3` indicates the models for up to 1, 2, and 3 elements are all verified. This new feature is left as future work.

7.1.3.4 Model Checking of Buffer Specification

When model-checking this case by ProB, we notice ProB kernel treats seq T as

$$set(couple(integer, T))$$

in Z and B. But it fails to match the sequence type in CSP. Therefore, it generates an incompatible type error. We change the implementation of predicate `type_ok` and `is_csp_set_type` in `specfile.pl` of ProB kernel source code to make

$$set(couple(integer, T))$$

Table 7.1: Model Checking Performance Comparison (Buffer Specification)

MAXINT	*MAXINS*	*maxbuff*	Time (*ms*)	Memory (MB)
3	3	1	122	38
3	3	2	538	38
3	3	3	2,152	39
3	4	4	28,022	40
3	5	5	443,306	78
1	9	9	16,802	68

match the sequence type in CSP. This modification has been accepted by the ProB team and merged into the source code of ProB.

Additionally, **div**, the most divergent process, is explicitly used in our link rules, such as Link Rule 31, Link Rule 53, Link Rule 55, and Link Rule 56, to capture the behaviour if the precondition of the schema in a schema expression as action is not satisfied, or none of guarded conditions in an alternation hold, or the precondition in a specification statement or an assumption does not hold. However **div** is not available in CSP$_M$ as well as ProB. Instead of defining an explicit process **div** via hiding, we define a non-divergent process DIV to correspond to **div**. The process DIV is defined as follows,

$$DIV = div \rightarrow STOP$$

where div is a special event in CSP. For example, **div** in Equation 7.5 becomes DIV. Though DIV is not a divergent process, we can check deadlock of combination of CSP and Z specifications to achieve divergence checking. We use the deadlock checking to find this kind of divergence because it is a more direct checking in ProB. In case that a deadlock is found, we check the counterexample to see if the last event is div or not. If the last event is div, it means the original *Circus* specification can lead to divergence. Alternatively, LTL formula checking can be used to check deadlock as well. For example, the LTL formula (`not F e(div)`), which denotes the statement that finally div event is enabled, is not true. When it comes to this case, if we remove guarded conditions in *Input* or *Output* action, the specification diverges because the precondition of *InputCmd* and *OutputCmd* may not hold. In the final CSP specification, the corresponding boolean guard is removed as well. Using ProB, we can easily find the deadlock and the last event is div, therefore it finds divergence. It is worth noting that with this deadlock checking of the div event we can only check divergences captured in our link rules. For other divergences such as the divergence caused by hiding events, they can be checked using CSP assertions like `assert Buffer :[ livelock free ]`.

Deadlock and Invariant Violation Checking Finally, we can model-check the combination of CSP and Z specifications and there is no deadlock found. A comparison of the model checking performance for different configuration of constants is shown in Table 7.1. This experiment was undertaken on ProB Linux version, which is modified based on ProB 1.5.0-Beta, on Ubuntu.

Deadlock and Divergence Checking by CSP Assertions ProB is capable of deadlock and divergence checking through CSP assertions as well. By adding the following three asserts to the CSP model, we checked the deadlock free and divergence free of the *Buffer* process with the combination of constants in Table 7.1 successfully.

```
assert Buffer :[ deadlock [F] ]
assert Buffer :[ deadlock [FD] ]
assert Buffer :[ livelock free ]
```

Table 7.2: Model Checking Performance Comparison (Buffer Implementation)

MAXINT	*maxbuff*	*maxring*	Time (*ms*)	Memory (MB)
3	2	1	38,039	53
3	3	2	2,582,944	837
1	4	3	951,593	913
1	4	3	236,532	318 [a]

[a]This row is the result with substituted initialisation schema.

7.1.3.5 Model Checking of Distributed Reactive Buffer

One issue we found is about the well-definedness of the modulo operation in Z when it is translated to the counterpart in B. In Z, the modulo operation is defined on the integer dividend and the non-zero integer divisor [4]. However it is defined on the natural number dividend and the non-zero natural number divisor in B machine. Therefore, when model-checking this case by ProB that translates the modulo to the modulo operation in B, it triggers an error about the well-definedness of the modulo because the dividend of the modulo in Z is possibly less than 0. Thus, we modified the implementation of the modulo operation in ProB to use the modulo operation *mod* in SICStus Prolog, and this modification has been accepted by the ProB team and merged into the latest source code of ProB. Because the modulo operation in Z uses truncation towards minus infinity [4] and in Prolog it is the integer remainder after floored division [93], they use the same definition of modulo—floored division [94]. Hence, the well-definedness of mod in Z is retained.

In addition, ProB uses the built-in command time in Tcl to measure the elapsed time for the model checking task. It can count up to 4,294,967,295 microseconds, approximately 72 minutes, for a task in a 32-bit machine, otherwise it will cause the overflow. For the model checking of this case with the *maxbuff* larger than 3, it requires longer time and causes the overflow. Therefore, the output result about the time is not useful. We record the timestamp before the task execution and the timestamp after the completion of the task by `clock milliseconds` in Tcl, then calculate the difference between two timestamps. This is the model checking time. This modification made to ProB is only for temporary test purpose.

Deadlock and Invariant Violation Checking There is no deadlock or divergence found. A comparison of the model checking performance is shown in Table 7.2. Note that due to the state space exploration and resource limitation, we are not able to model check this case if *maxbuff* is larger than 3 and *MAXINT* is 3 because ProB runs out of memory on Linux with 3 GB memory. We can set the *MAXINT* to 1 to reduce the state space. The result is shown in the third row. Further methods like more specific initialisation can be used to tremendously decrease the size of the state space. For an instance, if we substitute by the initialisation schema (7.8), the model checking result is displayed in the fourth row.

$$\begin{aligned} Init \mathrel{\widehat=} [\, State' \mid\ & Controller\_top' = 1 \land Controller\_bot' = 1 \\ & \land Controller\_size' = 0 \land Controller\_ringsize' = 0 \\ & \land Controller\_cache' = 0 \land RingCell\_v' = 0 \\ & \land IRCell\_1\_v' = 0 \land IRCell\_2\_v' = 0 \\ & \land IRCell\_3\_v' = 0 \land IRCell\_4\_v' = 0 \,] \end{aligned} \tag{7.8}$$

Deadlock and Divergence Checking by CSP Assertions By adding the following three asserts to the CSP model, we checked the deadlock free and divergence free of the *Buffer* process with the combination of constants in Table 7.2 successfully.

Table 7.3: Refinement Checking Performance

Model	*MAXINT*	*maxbuff*	*maxring*	Time (*ms*)
Traces	1	3	2	109,180
Failures	1	3	2	122,440
Traces	3	3	2	342,440
Failures	3	3	2	355,320

```
assert Buffer :[ deadlock [F] ]
assert Buffer :[ deadlock [FD] ]
assert Buffer :[ livelock free ]
```

7.1.3.6 Refinement Checking

In addition, ProB can check if an implementation in $CSP \parallel B$ is a trace refinement of a specification in $CSP \parallel B$ [95]. When checking the trace refinement, an issue we got in ProB for our case, after inspecting source code, is that ProB refinement checker compares the traces of both the specification and the implementation according to their transitions in the same state space. That works for the refinement of two processes in the same CSP program for the CSP model, or the refinement of two processes in the same CSP program for the $CSP \parallel B$ model. But for our case, the specification and the implementation have the different Z programs and it is impossible to put their CSP programs into one CSP file. Thus we modified the `refinement_checker.pl` of ProB to search the traces by the transitions from their own separate state space. This change works for the buffer case and the ESEL case. However it has not been adopted by the ProB team yet. After model-checking the buffer specification, we save its state space for later refinement check to a file. Then we load the buffer implementation to ProB, and select "trace refinement check" function, open the saved state space file for the specification. Finally ProB will show the result: the implementation is a trace refinement of the specification; or if not a trace refinement, a counter example is provided.

We checked the trace refinement between the buffer specification and the buffer implementation, and finally got the result the distributed reactive buffer is a trace refinement of the buffer specification for *MAXINT* and *maxbuff* equal to 3 and 3 separately. Furthermore, ProB has an option to check failures. We checked the failure refinement between the specification and the implementation as well, and finally found the distributed reactive buffer is also a failure refinement of the buffer specification with the same constants. The refinement checking performance is shown in Table 7.3. According to Section 7.1.3.2, we can conclude the buffer implementation is a failure refinement of the buffer specification.

However, if *maxbuff* between the specification and the implementation is not equal, ProB gives an error with a counterexample provided.

7.2 ESEL

7.2.1 Requirement

Electronic Shelf Edge Label (ESEL), a technology to allow retailer stores to use digital price tags to manage the price update of products, has been widely used to facilitate store management. The requirement of this example is to design a ESEL system which receives price information from the server and then automatically updates the price of all products to associated ESEL tags.

Furthermore, the problem described in this section is an abstract and simplified ESEL system.

- Firstly, only information showed on the display of tags is the price of products and the unit of price is natural number.
- Secondly, the communication to update tags is abstract regardless of specific network and media. Therefore, the communication problems such as package lose and unreachable tags because of out of communication range, are not considered.
- Thirdly, the map between products and tags has been determined in advance. The system just receives this map information.

7.2.1.1 Environment and Physical Units

The environment and physical units that interact with this system are displayed in Figure 7.6.

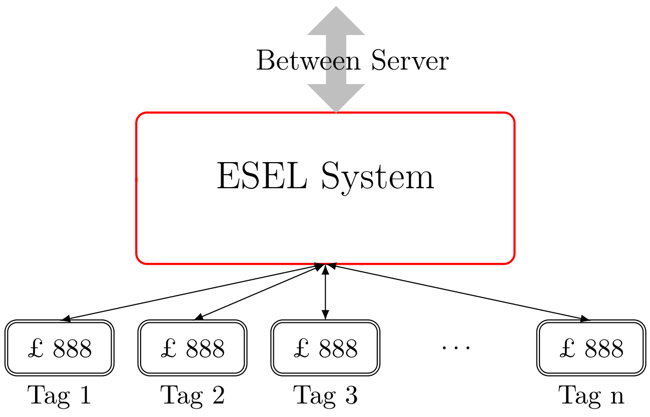

Figure 7.6: Environment of ESEL System

Server The system gets price information and the map between products and tags from the *server*, and after update it returns feedbacks to the *server*.

Tags or Displays Additionally, there are many *tags* or *displays* in the store to show the price of products and for each tag it is linked to up to one product. Here a *tag* is equal to a *display* and therefore the *display* will be used afterwards to denote a tag. Its functionalities are listed below.

- It does not support a communications network and is not able to communicate with other *displays*.
- It provides an interface to read or write the price data from or to the *display*. The data written to the *display* will be stored and shown on the screen if it is on. Even after the *display* is turned off, the data is still kept in its memory unless there is another write to override it. The read operation will return the stored value.
- It also provides another interface to turn on or off the screen of the *display*.

7.2.1.2 Functionalities of ESEL System

The ESEL system is designed to update the price of products to their corresponding displays.

- It receives the map between products and displays from the server and keeps it for further usage. It is capable of receiving either a partial or a full map. In a partial map, for all displays not in the new map, they are still the same—either map to existing products or not assigned. And for all displays in the new map, finally they override the stored map. In a full map, the stored map is discarded and the new map will be stored. In both modes, the new updated map will be kept and used in update cycles until there is another update of the map.
- The price map between products and price is very similar to the display map between displays and products. It provides partial or full update as well.
- An update command from the server initiates a price update cycle. The price information stored will be shown to associated displays. If the displays have problems that are identified through comparison of the data read from the displays with the data written to the displays, or the products do not have associated displays, error information will be returned to the server.
- In the initialisation stage, all displays are turned off.
- Each update cycle is a complete update and all displays are involved. For the displays that are not linked to any products or the displays whose associated products are not in the update list, they should be turned off.

7.2.2 Specification

The specification has been illustrated in Section 2.3.2, and the ESELHeader is given in Section 2.3.1. The complete *Circus* model is shown in Appendix H.2.1 and its linked CSP and Z models are in Appendix H.2.2.

7.2.3 System One

For the system displayed in Figure 2.1, considering the fact that all displays are scattered throughout the store, it is not wise to connect each display to the central controller directly through wires. A better solution is to have each display updated remotely without wire. Then they can be moved from one location to another easily. To support this remote update, displays should support wireless communication. Though data rich networks such as WiFi can make the communication between displays and the central controller easy, the cost of this solution with WiFi chips in displays is comparatively higher because of the large amount of displays. As a result, data-less communications such as ZigBee and RFID are commonly used for communication between the central controller and displays. Finally, the cost is highly reduced.

After this situation is taken into account, an additional controller is added to each display. The display and its controller together form an *ESEL* which is identified by an *ESEL ID* and this controller is consequently named the *ESEL controller*. In addition, this controller supports wireless connection and finally the ESELs can be updated remotely through the system controller. The solution is illustrated in Figure 7.7. Each unit U_i in the system controller is responsible for the update of one ESEL by communicating with the corresponding *ESEL controller*.

In the *System Controller*, the top three elements—*ESEL map*, *Price map*, and *Response*—keep the map between ESELs and products, the map between products and price, and the response respectively. The bottom n elements from U_1 to U_n are interleaved together, and each unit is responsible for the update of one ESEL by communicating with the corresponding *ESEL controller*. Furthermore, each unit can read the *ESEL map* and the *Price map* but will not update them, and the responses from the *ESEL controllers* are sent to the *Collector* which collects all responses and then updates the *Response*. With

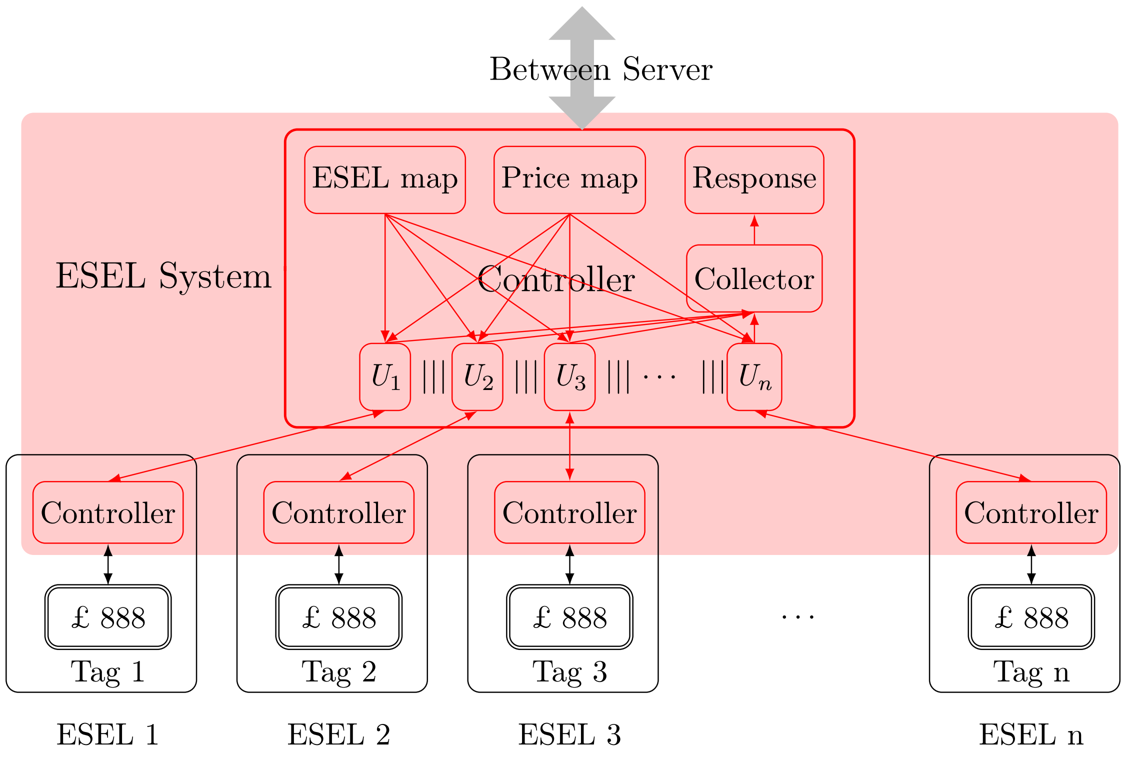

Figure 7.7: ESEL System One

this architecture, all ESELs can be updated at the same time because of interleaved update units ($U_1, .., U_n$), and the *Response* can be updated only by the *Collector* and therefore it is not necessary to introduce a lock.

7.2.3.1 *Circus* Model

This *Circus* model can also be found in Appendix H.3.1.

$$\textbf{section}\ \mathit{ESELSystem1}\ \textbf{parents}\ \mathit{ESELHeader}$$

Controller Process The process for the central controller, named *Controller*1, is defined as an explicitly defined process.

$$\textbf{process}\ \mathit{Controller1} \mathrel{\widehat=} \textbf{begin}$$

*Controller*1 has three state components: *pumap* for the map from ESELs to products, *ppmap* for the map from products to their price, and *response* for the response of one update cycle to the environment.

$$\textbf{state}\ \mathit{State} == [\, \mathit{pumap} : \mathit{ESID} \pfun \mathit{PID} ; \mathit{ppmap} : \mathit{PID} \pfun \mathit{Price}; \\ \mathit{response} : \mathit{PID} \pfun (\mathbf{P}\ \mathit{FStatus}) \,]$$

Initially, these three state components all are empty.

$$\mathit{Init} == [\, (\mathit{State})' \mid \mathit{pumap}' = \varnothing \land \mathit{ppmap}' = \varnothing \land \mathit{response}' = \varnothing \,]$$

The *UpdateMap* schema updates part of the map from ESELs to products according to the input map, while the *UpdateAllMap* schema discards stored map and uses new input map as *pumap*.

$$
\begin{aligned}
&\mathit{UpdateMap} == [\,\Delta \mathit{State}\,;\, \mathit{map?} : \mathit{ESID} \pfun \mathit{PID} \mid \\
&\qquad \mathit{pumap}' = \mathit{pumap} \oplus \mathit{map?} \land \mathit{ppmap}' = \mathit{ppmap} \land \mathit{response}' = \mathit{response}\,] \\
&\mathit{UpdateAllMap} == [\,\Delta \mathit{State}\,;\, \mathit{map?} : \mathit{ESID} \pfun \mathit{PID} \mid \\
&\qquad \mathit{pumap}' = \mathit{map?} \land \mathit{ppmap}' = \mathit{ppmap} \land \mathit{response}' = \mathit{response}\,]
\end{aligned}
$$

The *NewPrice* updates part of the stored map from products to price, while the *AllNewPrice* discards all price information stored and uses input price as *ppmap*.

$$
\begin{aligned}
&\mathit{NewPrice} == [\,\Delta \mathit{State}\,;\, \mathit{price?} : \mathit{PID} \pfun \mathit{Price} \mid \\
&\qquad \mathit{ppmap}' = \mathit{ppmap} \oplus \mathit{price?} \land \mathit{pumap}' = \mathit{pumap} \land \mathit{response}' = \mathit{response}\,] \\
&\mathit{AllNewPrice} == [\,\Delta \mathit{State}\,;\, \mathit{price?} : \mathit{PID} \pfun \mathit{Price} \mid \\
&\qquad \mathit{ppmap}' = \mathit{price?} \land \mathit{pumap}' = \mathit{pumap} \land \mathit{response}' = \mathit{response}\,]
\end{aligned}
$$

AUpdatemap is an action defined to update the map from ESELs to products: either partially by the *updatemap* event or completely by the *updateallmap* event.

$$
\begin{aligned}
&\mathit{AUpdatemap} \mathrel{\widehat=} \mathit{updatemap?map} \to \big(\mathit{UpdateMap}\big) \\
&\qquad \Box\; \mathit{updateallmap?map} \to \big(\mathit{UpdateAllMap}\big)
\end{aligned}
$$

Similarly, *ANewPrice* is an action defined to update the map from products to price: either partially by the *updateprice* event or completely by the *updateallprice* event.

$$
\begin{aligned}
&\mathit{ANewPrice} \mathrel{\widehat=} \mathit{updateprice?price} \to \big(\mathit{NewPrice}\big) \\
&\qquad \Box\; \mathit{updateallprice?price} \to \big(\mathit{AllNewPrice}\big)
\end{aligned}
$$

A parametrised action, *AUpdateUnitPrice*, is given to update the price (specified by the formal *pid* parameter) to an ESEL (given by the formal *uid* parameter). It sends the price to the specified ESEL by the *uupdate* event, and then waits for the response from the ESEL. If the return status is not successful (*ufail*), it sends the result to the response collection action *CollectResp* below, then terminates. Otherwise, it terminates immediately.

$$
\begin{aligned}
&\mathit{AUpdateUnitPrice} \mathrel{\widehat=} \mathit{uid} : \mathit{ESID}\,;\, \mathit{pid} : \mathit{PID} \bullet \\
&\qquad \mathit{uupdate.uid.}(\mathit{ppmap}\ \mathit{pid}) \to \mathit{ures.uid?rst} \to \\
&\qquad ((\mathit{rst} = \mathit{ufail})\ \&\ \mathit{resp.pid.}(\mathit{fail}\ \mathit{uid}) \to \mathbf{Skip} \\
&\qquad \Box\; (\mathit{rst} = \mathit{uok})\ \&\ \mathbf{Skip})
\end{aligned}
$$

The parametrised action *AUpdateProductUnits* aims to update one product's price specified by the formal *pid* parameter in case the product has associated ESELs. Since one product may have more than one associated ESELs, this action updates the product's price to all associated ESELs. Furthermore, the update to each ESEL is independent. Therefore, they are combined together into an interleave. It is worth noting that each *AUpdateUnitPrice* action will not update state or local variables and thus its name set is empty.

$$
\begin{aligned}
&\mathit{AUpdateProductUnits} \mathrel{\widehat=} \mathit{pid} : \mathit{PID} \bullet \\
&\qquad (\mathbin{|||}\ \mathit{uid} : (\operatorname{dom}(\mathit{pumap} \rhd \{\mathit{pid}\}))\ \llbracket \varnothing \rrbracket \bullet \mathit{AUpdateUnitPrice}(\mathit{uid}, \mathit{pid}))
\end{aligned}
$$

Otherwise, if the product has not been allocated with the corresponding ESELs, it sends back a response to state this error *NA*. The behaviour is defined in the *AUpdateNoUnit* action.

$$
\mathit{AUpdateNoUnit} \mathrel{\widehat=} \mathit{pid} : \mathit{PID} \bullet \mathit{resp.pid.NA} \to \mathbf{Skip}
$$

The behaviour of the price update for a product given in *pid* is the update of the product either with associated ESELs, guarded *AUpdateProductUnits*, or without associated ESELs, guarded *AUpdateNoUnit*.

$$\begin{aligned}&AUpdateProduct \mathrel{\widehat=} pid : PID \bullet\\ &\quad\quad (pid \in \operatorname{ran} pumap) \mathrel{\&} AUpdateProductUnits(pid)\\ &\quad\ \Box\ (pid \notin \operatorname{ran} pumap) \mathrel{\&} AUpdateNoUnit(pid)\end{aligned}$$

Then the update of all products is given in the action *AUpdateProducts*. Firstly, it is an interleave of all updates of the products which have associated price, then follows a *terminate* event to finish the update.

$$\begin{aligned}&AUpdateProducts \mathrel{\widehat=} ((\big|\big|\big|\ pid : (\operatorname{dom} ppmap) \,\|[\,\varnothing\,]\|\, \bullet AUpdateProduct(pid))\\ &\quad ; terminate \to \mathbf{Skip})\end{aligned}$$

The *AddOneFailure* schema below is used to add one product's failure status to *response*.

$$\begin{aligned}&AddOneFailure == [\,\Delta State\,;\,pid? : PID\,;\,fst? : FStatus \mid\\ &\quad (pid? \in \operatorname{dom} response \Rightarrow\\ &\quad\quad response' = response \oplus \{pid? \mapsto (response(pid?) \cup \{fst?\})\}) \land\\ &\quad (pid? \notin \operatorname{dom} response \Rightarrow\\ &\quad\quad response' = response \cup \{pid? \mapsto \{fst?\}\}) \land\\ &\quad ppmap' = ppmap \land pumap' = pumap\,]\end{aligned}$$

The *CollectResp* action is to collect responses from all units and write them into the *response* variable. It recursively waits for the response from the units, or terminates if required.

$$\begin{aligned}&CollectResp \mathrel{\widehat=} \mu X \bullet\\ &\quad ((resp?pid?fst \to \big(AddOneFailure\big)\,;\,X)\ \Box\ terminate \to \mathbf{Skip})\end{aligned}$$

Then the behaviour of the update of all products and the behaviour of response collection are put together into a *AUpdateResp* action. It is a parallel composition of the *AUpdateProducts* and *CollectResp* actions and they are synchronised on *resp* and *terminate* events. Finally, these internal events are hidden.

$$\begin{aligned}&AUpdateResp \mathrel{\widehat=}\\ &\quad (AUpdateProducts \,[\![\, \varnothing \mid RespInterface \mid \{response\} \,]\!]\, CollectResp)\\ &\quad \setminus RespInterface\end{aligned}$$

The overall price update action is given in *AUpdatePrice*, which accepts an *update* event from its environment, then clears the *response*, updates the price, sends the *display* event to make all ESELs show their price at the same time, then returns the response back to the environment.

$$\begin{aligned}&AUpdatePrice \mathrel{\widehat=} update \to response := \varnothing;\\ &\quad AUpdateResp\,;\,display \to finishdisplay \to failures.response \to \mathbf{Skip}\end{aligned}$$

The overall behaviour of the *Controller* process is given by its main action. It is initialised first, then repeatedly provides the ESEL map update, the price map update, or the price update to its environment.

$$\begin{aligned}&\bullet \big(Init\big)\,;\,init \to \mathbf{Skip};\\ &\quad (\mu X \bullet (AUpdatemap\ \Box\ ANewPrice\ \Box\ AUpdatePrice)\,;\,X)\end{aligned}$$

end

ESEL Process Each ESEL is defined as a parametrised process with the formal parameter *eid*.

$$\textbf{process}\ \ ESEL1 \mathrel{\widehat=} eid : ESID \bullet \textbf{begin}$$

The process has two state components: *price* for the price to display, and *status* for the status of ESEL.

$$\textbf{state}\ \ State == [\, price : Price \,;\, status : UStatus \,]$$

Initially, the price is equal to 0 and the status is *uok*.

$$Init == [\,(State)' \mid price' = 0 \land status' = uok\,]$$

The *Update* action provides its environment (the *Controller*) the update of price for the associated product. It accepts the *uupdate* event with the price, then writes the price to the *price* variable. After that, it writes the price to the display unit, and reads back the value to be compared with the original price. If they are equal, it sends back status *uok* by the *ures* event. Otherwise, it sends back status *ufail*. Accordingly, the *status* is updated.

$$\begin{aligned}&Update \mathrel{\widehat=} uupdate.eid?x \rightarrow price := x \,;\, write.eid.price \rightarrow read.eid?y\\&\quad\rightarrow ((y = price) \mathbin{\&} ures.eid.uok \rightarrow status := uok\\&\qquad\Box\ (y \neq price) \mathbin{\&} ures.eid.ufail \rightarrow status := ufail)\end{aligned}$$

The *Display* action accepts the *display* event. If the status is *uok*, then the associated display is turned on. Otherwise, the display is turned off.

$$\begin{aligned}&Display \mathrel{\widehat=} display \rightarrow (\\&\qquad (status = uok) \mathbin{\&} ondisplay.eid \rightarrow \textbf{Skip}\\&\quad\ \Box\ (status = ufail) \mathbin{\&} offdisplay.eid \rightarrow \textbf{Skip})\\&\quad\ ;finishdisplay \rightarrow \textbf{Skip}\end{aligned}$$

NotUpdateDisplay is an action to turn off this ESEL's display if it is not going to be updated.

$$NotUpdateDisplay \mathrel{\widehat=} display \rightarrow offdisplay.eid \rightarrow finishdisplay \rightarrow \textbf{Skip}$$

The initial behaviour of the process is given in the action *AInit* which initialises the state first, and then turns off the display.

$$AInit \mathrel{\widehat=} \big(Init\big) \,;\, offdisplay.eid \rightarrow init \rightarrow \textbf{Skip}$$

The overall behaviour of the process is given by its main action. It specifies that after initialisation the process repeatedly provides the update or the display to its environment.

$$\bullet\ AInit \,;\, (\mu X \bullet ((Update \,;\, Display) \ \Box\ NotUpdateDisplay) \,;\, X)$$

$$\textbf{end}$$

The behaviour of all ESELs together is formed by iterated parallel composition of the *ESEL* process. The communication between them is to synchronise on the events for initialisation and display.

$$\begin{aligned}&\textbf{channelset}\ \ InterESELInterface1 == \{\!|\, init, display, finishdisplay \,|\!\}\\&\textbf{process}\ \ ESELS1 \mathrel{\widehat=} \|\ eid : ESID \ [\![\, InterESELInterface1 \,]\!] \bullet ESEL1(eid)\end{aligned}$$

System Finally, the whole system is defined as the parallel composition between the *Controller* process and the *ESELS*1 process. They synchronise on the *uupdate*, *ures*, *init*, *display*, and *finishdisplay* events.

$$\textbf{channelset}\ ESELInterface1 == \{\!|\ uupdate, ures, init, display, finishdisplay\ |\!\}$$
$$\textbf{process}\ ESELSystem1 \mathrel{\widehat=} (Controller1 \llbracket ESELInterface1 \rrbracket ESELS1) \setminus ESELInterface1$$

7.2.3.2 CSP and Z Model

The result $CSP \parallel_B Z$ model is displayed in Appendix H.3.2.

7.2.4 System Two

For the system illustrated in Figure 7.7, considering a large store that has a big space and many ESELs, the usage of this central system controller has some disadvantages. Firstly, it is difficult to have full coverage of the wireless communication network from the central controller to each ESEL when the area is large. Secondly, a large amount of ESELs requires many units in the central control updating at the same time to maintain a reasonable overall update time. However, since all these updates are by wireless and these units in central controller are close to each other, then many units in the central controller working together will cause the wireless interfere problem.

To address these issues, our solution is to add several gateways between the central controller and ESELs. And each gateway is responsible for the update of registered ESELs in one area of the store. The main work of the gateways is to get the price information from the central controller, update all prices to their registered ESELs, collect update results, and finally send back these results to the central controller. And each gateway is individually not close in location and works independently. Therefore, the wireless interfere problem is alleviated, and update efficiency is improved because all gateways can update ESELs at the same time. This solution has relieved the central controller because the controller will not update ESELs directly. Instead, it only sends the price information to gateways and waits for the result. Consequently, the central controller is renamed to the *ESEL server*. The solution is illustrated in Figure 7.8.

7.2.4.1 *Circus* Model

This *Circus* model can also be found in Appendix H.4.1.

$$\textbf{section}\ ESELSystem2\ \textbf{parents}\ ESELHeader$$

An additional constant is defined. *MAX_GATEWAY* stands for maximum number of gateways in the system.

$$MAX\_GATEWAY : \mathbb{N}$$

Then all gateways are identified by *GID*. For instance, the number two is given $GW\,2$ or $GW\,(2)$.

$$GID ::= GW \langle\!\langle 1 \,..\, MAX\_GATEWAY \rangle\!\rangle$$

The map from ESELs to gateways, *gwmap*, is defined as a total function. One ESEL is linked to up to one gateway. However, one gateway may associate with multiple ESELs. For example,

$$gwmap : ESID \rightarrow GID$$
$$gwmap = \{(ES\,1, GW\,1), (ES\,2, GW\,1), (ES\,3, GW\,2)\}$$

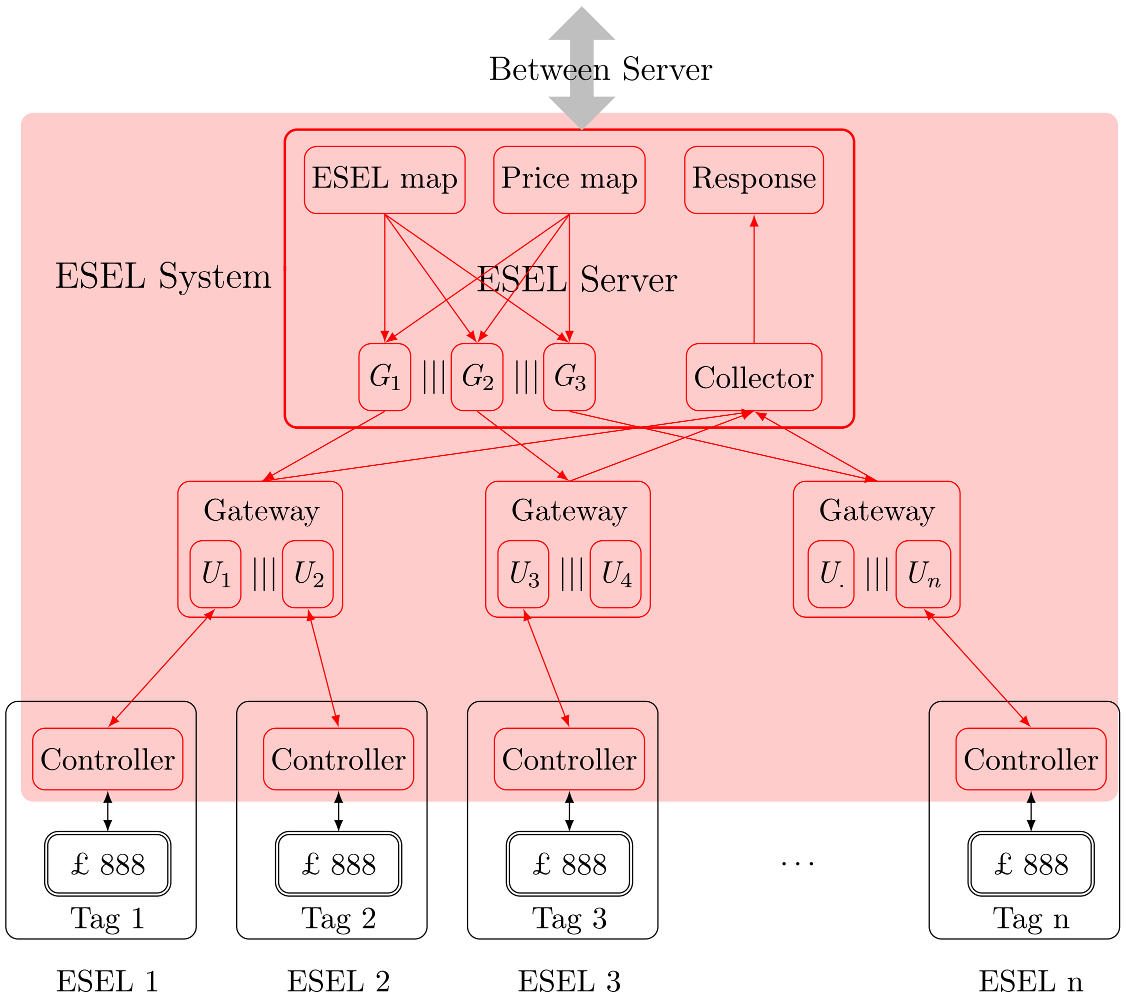

Figure 7.8: ESEL System Two

The channels below are used to communicate between the server and gateways, or within gateway internals. The server uses *gupdateprice* to send price information with ESEL IDs to the corresponding gateway, while *gfailure* is used to get back the update result from the gateway.

$$\textbf{channel}\ \ gupdateprice : GID \times (ESID \nrightarrow Price)$$
$$\textbf{channel}\ \ gfailure : GID \times \mathbf{P}\ ESID$$

gresp and *gterminate* are used in the internal of gateways to collect update results from each ESEL and terminate after collection.

$$\textbf{channel}\ \ gresp : ESID$$
$$\textbf{channel}\ \ gterminate$$
$$\textbf{channelset}\ \ GRespInterface == \{\!|\ gresp, gterminate\ |\!\}$$

ESEL Server Process The process for the ESEL server, named *ESELServer*, is defined as an explicitly defined process.

$$\textbf{process}\ \ ESELServer \mathrel{\widehat=} \textbf{begin}$$

The following definitions are the same as those in the *Controller*1. So their description is omitted.

$$\begin{array}{l}\textbf{state}\ \ State == [\, pumap : ESID \nrightarrow PID \,;\, ppmap : PID \nrightarrow Price ; \\ \quad response : PID \nrightarrow (\mathbb{P}\ FStatus) \,]\end{array}$$

$$Init == [\, (State)' \mid pumap' = \varnothing \land ppmap' = \varnothing \land response' = \varnothing \,]$$

$$\begin{array}{l}UpdateMap == [\, \Delta State \,;\, map? : ESID \nrightarrow PID \mid \\ \quad pumap' = pumap \oplus map? \land ppmap' = ppmap \land response' = response \,] \\ UpdateAllMap == [\, \Delta State \,;\, map? : ESID \nrightarrow PID \mid \\ \quad pumap' = map? \land ppmap' = ppmap \land response' = response \,]\end{array}$$

$$\begin{array}{l}NewPrice == [\, \Delta State \,;\, price? : PID \nrightarrow Price \mid \\ \quad ppmap' = ppmap \oplus price? \land pumap' = pumap \land response' = response \,] \\ AllNewPrice == [\, \Delta State \,;\, price? : PID \nrightarrow Price \mid \\ \quad ppmap' = price? \land pumap' = pumap \land response' = response \,]\end{array}$$

$$\begin{array}{l}AUpdatemap \mathrel{\widehat=} updatemap?map \rightarrow \big(UpdateMap \big) \\ \quad \Box\ updateallmap?map \rightarrow \big(UpdateAllMap \big)\end{array}$$

$$\begin{array}{l}ANewPrice \mathrel{\widehat=} updateprice?price \rightarrow \big(NewPrice \big) \\ \quad \Box\ updateallprice?price \rightarrow \big(AllNewPrice \big)\end{array}$$

$$AUpdateUnitFail \mathrel{\widehat=} eid : ESID \bullet resp.(pumap(eid)).(fail\ eid) \rightarrow \textbf{Skip}$$

$$AUpdateNoUnit \mathrel{\widehat=} pid : PID \bullet resp.pid.NA \rightarrow \textbf{Skip}$$

$$\begin{array}{l}ARespNoUnit \mathrel{\widehat=} \big|\big|\big|\ pid : (\mathrm{dom}\ ppmap \setminus \mathrm{ran}\ pumap) \ [\![\, \varnothing \,]\!] \bullet \\ \quad AUpdateNoUnit(pid)\end{array}$$

For each gateway, *AUpdateGateways* sends all price for the ESELs which are linked to the gateway and gets back update result. Then for each failure, the action passes it to *AUpdateUnitFail*, and finally writes to the *response*.

$$\begin{array}{l}AUpdateGateway \mathrel{\widehat=} gid : GID \bullet \\ \quad gupdateprice.gid!((\mathrm{dom}\,(gwmap \rhd \{gid\})) \lhd (pumap \mathbin{⨾} ppmap)) \rightarrow \\ \quad gfailure.gid?uids \rightarrow (\big|\big|\big|\ uid : uids \ [\![\, \varnothing \,]\!] \bullet AUpdateUnitFail(uid))\end{array}$$

The update of price to ESELs is an interleave of *AUpdateGateway* for all gateways.

$$AUpdateGateways \mathrel{\widehat=} \big|\big|\big|\ gid : GID \ [\![\, \varnothing \,]\!] \bullet AUpdateGateway(gid)$$

Then the update of all products, given in the action *AUpdateProducts*, is the interleave of the update of price to ESELs through gateways and the action for the case without associate ESELs. Then it follows a *terminate* event to finish the update.

$$\begin{array}{l}AUpdateProducts \mathrel{\widehat=} (AUpdateGateways \ [\![\, \varnothing \mid \varnothing \,]\!] \ ARespNoUnit); \\ \quad terminate \rightarrow \textbf{Skip}\end{array}$$

$$\begin{array}{l} \mathit{AddOneFailure} == [\Delta \mathit{State} ; \mathit{pid?} : \mathit{PID} ; \mathit{fst?} : \mathit{FStatus} \mid \\ \quad (\mathit{pid?} \in \operatorname{dom} \mathit{response} \Rightarrow \\ \qquad \mathit{response}' = \mathit{response} \oplus \{\mathit{pid?} \mapsto (\mathit{response}(\mathit{pid?}) \cup \{\mathit{fst?}\})\}) \land \\ \quad (\mathit{pid?} \notin \operatorname{dom} \mathit{response} \Rightarrow \\ \qquad \mathit{response}' = \mathit{response} \cup \{\mathit{pid?} \mapsto \{\mathit{fst?}\}\}) \land \\ \quad \mathit{ppmap}' = \mathit{ppmap} \land \mathit{pumap}' = \mathit{pumap}] \end{array}$$

$$\begin{array}{l} \mathit{ACollectResp} \mathrel{\widehat=} \mu X \bullet \\ \quad ((\mathit{resp?pid?fst} \to \big(\mathit{AddOneFailure}\big) ; X) \mathrel{\Box} \mathit{terminate} \to \mathbf{Skip}) \end{array}$$

Then the update of all products and the response collection behaviours are put together into the *AUpdateResp* action. It is a parallel composition of the *AUpdateProducts* and *CollectResp* actions and they are synchronised with *resp* and *terminate* events. Finally, these internal events are hidden.

$$\begin{array}{l} \mathit{AUpdateResp} \mathrel{\widehat=} \\ \quad (\mathit{AUpdateProducts} \llbracket \varnothing \mid \mathit{RespInterface} \mid \{\mathit{response}\} \rrbracket \mathit{ACollectResp}) \\ \quad \setminus \mathit{RespInterface} \end{array}$$

The overall price update action is given in *AUpdatePrice*, which accepts an *update* event from its environment, then clears *response*, updates the price, sends the *display* event to make all ESELs show their price at the same time, then feeds back the response to the environment.

$$\begin{array}{l} \mathit{AUpdatePrice} \mathrel{\widehat=} \mathit{update} \to \mathit{response} := \varnothing ; \\ \quad \mathit{AUpdateResp} ; \mathit{display} \to \mathit{finishdisplay} \to \mathit{failures.response} \to \mathbf{Skip} \end{array}$$

The overall behaviour of the *ESELServer* process is given by its main action. It initializes first, then repeatedly provides the ESEL map update, the price map, or the price update to its environment.

$$\begin{array}{l} \bullet \big(\mathit{Init}\big) ; \mathit{init} \to \mathit{finishinit} \to \mathbf{Skip} ; \\ \quad (\mu X \bullet (\mathit{AUpdatemap} \mathrel{\Box} \mathit{ANewPrice} \mathrel{\Box} \mathit{AUpdatePrice}) ; X) \end{array}$$

end

Gateway Process The *Gateway* process is defined as a parametrised process.

$$\mathbf{process}\ \mathit{Gateway} \mathrel{\widehat=} \mathit{gid} : \mathit{GID} \bullet \mathbf{begin}$$

It has two state components: *pumap* for the map from ESELs to price, and *failed* for a set of ESELs which update unsuccessfully.

$$\mathbf{state}\ \mathit{State} == [\mathit{pumap} : \mathit{ESID} \pfun \mathit{Price} ; \mathit{failed} : \mathbb{P}\ \mathit{ESID}]$$

Initially, both are empty.

$$\mathit{Init} == [(\mathit{State})' \mid \mathit{pumap}' = \varnothing \land \mathit{failed}' = \varnothing]$$

The map can be updated only completely and cannot be updated partially.

$$\begin{array}{l} \mathit{UpdateAllMap} == [\Delta \mathit{State} ; \mathit{map?} : \mathit{ESID} \pfun \mathit{Price} \mid \\ \quad \mathit{pumap}' = \mathit{map?} \land \mathit{failed}' = \mathit{failed}] \end{array}$$

The map is updated after the input from *ESELServer* through the *gupdateprice* channel.

$$AUpdateallmap \mathrel{\widehat=} gupdateprice.gid?map \rightarrow \big(UpdateAllMap\big)$$

A parametrised action, *AUpdateUnitPrice*, is given to update the price to an ESEL (given by the formal *uid* parameter). It sends the price to the specific ESEL by the *uupdate* event, and then waits for the response from the ESEL. If the return status is not successful (*ufail*), it sends the result to the response collection action *CollectResp* below, then terminates. Otherwise, it terminates immediately.

$$\begin{aligned}&AUpdateUnitPrice \mathrel{\widehat=} uid : ESID \bullet\\ &\quad uupdate.uid.(pumap\ uid) \rightarrow ures.uid?rst \rightarrow\\ &\quad ((rst = ufail)\ \&\ gresp!uid \rightarrow \mathbf{Skip}\\ &\quad \Box\ (rst = uok)\ \&\ \mathbf{Skip})\end{aligned}$$

The updates of all ESELs in this gateway are put in an iterated interleave, then followed a *gterminate* event to finish the updates.

$$\begin{aligned}&AUpdateAllUnits \mathrel{\widehat=} ((\,|||\ eid : (\mathrm{dom}\ pumap) \parallel\![\ \varnothing\ ]\!\parallel \bullet AUpdateUnitPrice(eid))\\ &\quad ; gterminate \rightarrow \mathbf{Skip})\end{aligned}$$

The *CollectResp* action is to collect responses from all units and write them into the *response* variable. It recursively waits for the response from the units, or terminates if required.

$$\begin{aligned}&AGCollectResp \mathrel{\widehat=} \mu X \bullet\\ &\quad ((gresp?uid \rightarrow failed := failed \cup \{uid\}\ ;\ X)\ \Box\ gterminate \rightarrow \mathbf{Skip})\end{aligned}$$

Then the updates of all products and the response collection are put together into the *AUpdateResp* action. It is a parallel composition of the *AUpdateProducts* and *CollectResp* actions and they are synchronised on the *resp* and *terminate* events. Finally, these internal events are hidden.

$$\begin{aligned}&AGUpdateResp \mathrel{\widehat=}\\ &\quad (AUpdateAllUnits\ [\![\ \varnothing \mid GRespInterface \mid \{failed\}\ ]\!]\ AGCollectResp)\\ &\quad \setminus GRespInterface\end{aligned}$$

The overall price update action is given in *AUpdatePrice*, which accepts a *gupdateprice* event from its environment, then clears *failed*, updates the price, sends update results to the server, and waits for the *display* event to make all ESELs in this gateway show their price at the same time.

$$\begin{aligned}&AGUpdatePrice \mathrel{\widehat=} AUpdateallmap\ ;\ failed := \varnothing;\\ &\quad AGUpdateResp\ ;\ gfailure.gid!failed \rightarrow display \rightarrow udisplay \rightarrow\\ &\quad finishudisplay \rightarrow finishdisplay \rightarrow \mathbf{Skip}\end{aligned}$$

The overall behaviour of the *Gateway* process is given by its main action. It is initialised first, then repeatedly provides price update to its environment.

$$\begin{aligned}&\bullet \big(Init\big)\ ;\ init \rightarrow uinit \rightarrow finishuinit \rightarrow finishinit \rightarrow \mathbf{Skip};\\ &\quad (\mu X \bullet (AGUpdatePrice)\ ;\ X)\end{aligned}$$

end

ESEL Process Each ESEL is defined as a parametrised process with the formal parameter *eid*.

$$\textbf{process}\ \ ESEL2 \mathrel{\widehat=} eid : ESID \bullet \textbf{begin}$$

The process has two state components: *price* for the price to display, and *status* for the status of ESEL.

$$\textbf{state}\ \ State == [\, price : Price \,;\, status : UStatus \,]$$

Initially, the price is equal to 0 and the status is *uok*.

$$Init == [\, (State)' \mid price' = 0 \land status' = uok \,]$$

The *Update* action provides its environment (*Gateway*) the update of price for the associated product. It accepts the *uupdate* event with the price, then writes the price to *price*. After that, it writes the price to the display unit, and reads back the value to be compared with the original price. If they are equal, it sends back status *uok* by the *ures* event. Otherwise, it sends back status *ufail*. Accordingly, *status* is updated.

$$\begin{aligned}
&Update \mathrel{\widehat=} uupdate.eid?x \rightarrow price := x \,;\, write.eid.price \rightarrow read.eid?y \\
&\quad \rightarrow ((y = price) \,\&\, ures.eid.uok \rightarrow status := uok \\
&\qquad \Box\ (y \neq price) \,\&\, ures.eid.ufail \rightarrow status := ufail)
\end{aligned}$$

The *Display* action accepts the *udisplay* event. If the status is *uok*, then the associated display is turned on. Otherwise, the display is turned off.

$$\begin{aligned}
&Display \mathrel{\widehat=} udisplay \rightarrow (\\
&\qquad (status = uok) \,\&\, ondisplay.eid \rightarrow \textbf{Skip} \\
&\quad\ \Box\ (status = ufail) \,\&\, offdisplay.eid \rightarrow \textbf{Skip}) \\
&\quad\ ; finishudisplay \rightarrow \textbf{Skip}
\end{aligned}$$

$$NotUpdateDisplay \mathrel{\widehat=} udisplay \rightarrow offdisplay.eid \rightarrow finishudisplay \rightarrow \textbf{Skip}$$

The initial behaviour of the process is given in the action *AInit* which initialises the state at first, and then turns off the display.

$$AInit \mathrel{\widehat=} \big(Init\big) \,;\, uinit \rightarrow offdisplay.eid \rightarrow finishuinit \rightarrow \textbf{Skip}$$

The overall behaviour of the process is given by its main action. It specifies that after initialisation the process repeatedly provides the update or the display to its environment.

$$\bullet\ AInit \,;\, (\mu X \bullet ((Update \,;\, Display) \,\Box\, NotUpdateDisplay) \,;\, X)$$

$$\textbf{end}$$

System All ESELS which are registered with the same gateway synchronise on the unit initialisation and display events.

$$\begin{aligned}
&\textbf{channelset}\ \ InterESELInterface2 == \{\!|\, uinit, finishuinit, \\
&\qquad\qquad\qquad\qquad udisplay, finishudisplay \,|\!\} \\
&\textbf{process}\ \ ESELS2 \mathrel{\widehat=} gid : GID \bullet \\
&\quad (\,\|\ eid : (\mathrm{dom}\,(gwmap \rhd \{gid\}))\ [\![\, InterESELInterface2]\!] \bullet ESEL2(eid))
\end{aligned}$$

Each gateway is put in parallel with its linked ESELs and all gateways synchronise on the gateway initialisation and display events.

$$\begin{array}{l}
\textbf{channelset}\ \ InterGWInterface2 == \{\!|\ init, finishinit, display, finishdisplay\ |\!\} \\
\textbf{channelset}\ \ GWESELInterface2 == \{\!|\ uinit, finishuinit, uupdate, ures, \\
\qquad\qquad\qquad udisplay, finishudisplay\ |\!\} \\
\textbf{process}\ \ Gateways \mathrel{\widehat=} \|\ gid : GID\ [\![\ InterGWInterface2]\!] \bullet \\
\qquad (Gateway(gid)\ [\![\ GWESELInterface2\ ]\!]\ ESELS2(gid)) \setminus GWESELInterface2
\end{array}$$

Finally, the ESEL System Two is simply the parallel composition of the *ESELServer* and the *Gateways*, and communications between them are hidden.

$$\begin{array}{l}
\textbf{channelset}\ \ ServerGWInterface == \{\!|\ init, finishinit, gupdateprice, gfailure, \\
\qquad\qquad\qquad display, finishdisplay\ |\!\} \\
\textbf{process}\ \ ESELSystem2 \mathrel{\widehat=} \\
\qquad (ESELServer\ [\![\ ServerGWInterface\ ]\!]\ Gateways) \setminus ServerGWInterface
\end{array}$$

7.2.4.2 CSP and Z Model

The result $CSP \parallel_B Z$ model is displayed in Appendix H.4.2.

7.2.5 Model Checking Results

7.2.5.1 Deadlock and Livelock Checking

According to Section 6.5.2, our approach can model-check only one instance of all constants each time. Table 7.4 shows the model-checking results for some configurations of constants. The first four columns denote the specific configuration, followed by resources consumption (time and memory), and the last three columns give the number of states and transitions checked and corresponding percentages. To obtain these checking results, ProB runs on a server having 512GB RAM, 32-core CPU, and running Ubuntu 14.04.4 OS. That only small constants have been checked (and some are not completely checked) is due to the fact that ProB runs on only one process, or is not able to support multiple threads and processes currently because of the limitations of SICStus Prolog [83], the logic programming language for the kernel of ProB. In the table, if the percentage of checked states is 100, it means the configuration is checked to be deadlock free and livelock free. Otherwise, if the percentage is less than 100, then it states that the configuration has been partially checked and no deadlock and livelock found.

7.2.5.2 Refinement Checking

With ProB, we can check refinement in terms of the traces model and the failures model for $CSP \parallel B$. For a configuration of constants, we have checked the System One is a refinement of the Specification and the System Two is a refinement of the System One in terms of both models. The results are shown in Table 7.4.

7.2.5.3 Properties Checked

P1: All ESELs are displayed together In case of the display events (*ondisplay* or *offdisplay*), all ESELs should be engaged. To check this property, we add one *Circus* process *AllDisplayChecker* to simulate the system's environment.

$$\textbf{section}\ \ ESELAllDisplayChecker\ \textbf{parents}\ \ ESELHeader$$

$$\textbf{process}\ \ AllDisplayChecker \mathrel{\widehat=} \textbf{begin}$$

Table 7.4: ESEL Model Checking Results

S [1]	P [2]	W [3]	INT[4]	Time (mins)	Memory (GB)	States	Transitions	% Checked
ESEL Specification								
2	2	-	1	7	0.26	61,968	127,232	100
2	2	-	2	25	0.73	151,872	312,896	100
3	2	-	1	644	9.50	1,021,793	2,133,145	100
3	3	-	1	6,856	149.10	55,024,795	110,063,861	100
ESEL System One								
2	2	-	1	100	3.07	565,194	1,166,765	100
2	2	-	2	511	13.98	2,735,905	5,546,214	100
3	2	-	1	1,399	64.90	5,795,002	19,283,900	53.6
3	3	-	1	809	36.80	1,487,002	8,777,035	23.6
ESEL System Two								
2	2	2	1	969	31.10	2,061,913	8,940,728	100
2	2	2	2	2,200	95.90	9,928,002	27,005,867	100
3	2	2	1	2,075	121.30	6,918,002	19,756,749	62.0
3	3	2	1	5,057	224.14	9,126,505	50,793,610	34.2
Refinement Checking								
2	2	2	1	$Spec \sqsubseteq_{T} SystemOne \sqsubseteq_{T} SystemTwo$				
2	2	2	1	$Spec \sqsubseteq_{F} SystemOne \sqsubseteq_{F} SystemTwo$				
Properties Checking								
2	2	2	1	$P1$, $P2$, and $P3$ checked for three models				
3	3	2	1	$P2$ and $P3$ checked for three models				

[1] MAX_ESEL [2] MAX_PID [3] $MAX\_GATEWAY$ [4] $MAXINT$

$$\begin{array}{l} \textbf{state}\ State == [\, dummy : \{0\} \,] \\ Init == [\, (State)' \mid dummy' = 0 \,] \end{array}$$

$$\begin{array}{l} ADisplay \mathrel{\widehat=} eid : ESID \bullet \\ \qquad (offdisplay.eid \to \mathbf{Skip} \,\Box\, ondisplay.eid \to \mathbf{Skip}) \\ AAllDisplay \mathrel{\widehat=} (\,|||\, e : ESID \bullet ADisplay(e)) \end{array}$$

$$\begin{array}{l} ACheck \mathrel{\widehat=} (AAllDisplay \,\Box\, updateprice?x \to \mathbf{Skip} \,\Box \\ \qquad updateprice?x \to \mathbf{Skip} \,\Box\, updatemap?x \to \mathbf{Skip} \,\Box \\ \qquad updatemap?x \to \mathbf{Skip} \,\Box\, update \to \mathbf{Skip} \,\Box \\ \qquad write?x \to \mathbf{Skip} \,\Box\, read?x \to \mathbf{Skip} \,\Box\, failures?x \to \mathbf{Skip}) \end{array}$$

$$\bullet\ \big(Init\big)\ ;\ \mu X \bullet (ACheck\ ;\ X)$$

$$\mathbf{end}$$

$$\begin{array}{l} \textbf{channelset}\ ESELSystemInterface == \{\!|\, updateallprice, updateprice, \\ \qquad updatemap, updateallmap, update, ondisplay, offdisplay, \\ \qquad write, read, failures \,|\!\} \end{array}$$

This process actually is an external choice of all system's external events except displays events. For the displays events, all ESELs are interleaved together and finally put in the external choice of other events. Eventually this process is in parallel with the model under test.

For the ESEL specification,

$\mathbf{section}\ \ ESELSpecAllDisplayChecker\ \mathbf{parents}\ \ ESELAllDisplayChecker, ESELSpec$

$\mathbf{process}\ \ ESELSpecAllDisplayChecker \mathrel{\widehat=}$
$\quad (AllDisplayChecker \llbracket ESELSystemInterface \rrbracket ESELSpec)$

For the System One,

$\mathbf{section}\ \ ESELSystem1AllDisplayChecker\ \mathbf{parents}$
$\quad ESELAllDisplayChecker, ESELSystem1$

$\mathbf{process}\ \ ESELSystem1AllDisplayChecker \mathrel{\widehat=}$
$\quad (AllDisplayChecker \llbracket ESELSystemInterface \rrbracket ESELSystem1)$

For the System Two,

$\mathbf{section}\ \ ESELSystem2AllDisplayChecker\ \mathbf{parents}$
$\quad ESELAllDisplayChecker, ESELSystem2$

$\mathbf{process}\ \ ESELSystem2AllDisplayChecker \mathrel{\widehat=}$
$\quad (AllDisplayChecker \llbracket ESELSystemInterface \rrbracket ESELSystem2)$

These new processes *ESELSpecAllDisplayChecker*, *ESELSystem1AllDisplayChecker*, and *ESELSystem2AllDisplayChecker* state that for displays events all ESELs shall engage together. And we use deadlock checking of ProB to check if this property holds for three models. The results are shown in Table 7.4.

P2: Price update only available after initialisation The *update* event is enabled only after initialisation and all displays are turned off as well. Our solution is to add an *InitChecker* process in *Circus* to simulate the system's environment.

$\mathbf{section}\ \ ESELInitChecker\ \mathbf{parents}\ \ ESELHeader$

$\mathbf{process}\ \ InitChecker \mathrel{\widehat=} \mathbf{begin}$
$\quad \mathbf{state}\ \ State == [\, dummy : \{0\} \,]$
$\quad Init == [\, (State)' \mid dummy' = 0 \,]$
$\quad AOffDisplay \mathrel{\widehat=} (\interleave\, e : ESID \bullet (offdisplay.e \rightarrow \mathbf{Skip}))$
$\quad \bullet \big(Init\big) \mathbin{;} AOffDisplay \mathbin{;} update \rightarrow \mathbf{Skip}$
$\mathbf{end}$

$\mathbf{channelset}\ \ ESELSystemInterface == \{\!| \, updateallprice, updateprice,$
$\qquad updatemap, updateallmap, update, ondisplay, offdisplay,$
$\qquad write, read, failures \,|\!\}$

This process begins with an interleave of the *offdisplay* event for all ESELs and after that it is engaged in an *update* event. Then this process is in parallel with the model under test on all external events of the system.

For the Specification,

$$\begin{array}{l}
\textbf{section}\ \ \mathit{ESELSpecInitChecker}\ \textbf{parents}\ \ \mathit{ESELInitChecker}, \mathit{ESELSpec}
\end{array}$$

$$\begin{array}{l}
\textbf{process}\ \ \mathit{ESELSpecInitChecker} \mathrel{\widehat=} \\
\qquad (\mathit{InitChecker} \llbracket \mathit{ESELSystemInterface} \rrbracket \mathit{ESELSpec})
\end{array}$$

For the System One,

$$\begin{array}{l}
\textbf{section}\ \ \mathit{ESELSystem1InitChecker}\ \textbf{parents}\ \ \mathit{ESELInitChecker}, \mathit{ESELSystem1}
\end{array}$$

$$\begin{array}{l}
\textbf{process}\ \ \mathit{ESELSystem1InitChecker} \mathrel{\widehat=} \\
\qquad (\mathit{InitChecker} \llbracket \mathit{ESELSystemInterface} \rrbracket \mathit{ESELSystem1})
\end{array}$$

For the System Two,

$$\begin{array}{l}
\textbf{section}\ \ \mathit{ESELSystem2InitChecker}\ \textbf{parents}\ \ \mathit{ESELInitChecker}, \mathit{ESELSystem2}
\end{array}$$

$$\begin{array}{l}
\textbf{process}\ \ \mathit{ESELSystem2InitChecker} \mathrel{\widehat=} \\
\qquad (\mathit{InitChecker} \llbracket \mathit{ESELSystemInterface} \rrbracket \mathit{ESELSystem2})
\end{array}$$

Finally, we use ProB's temporal logic checking of this formula (`F e(update)`) which means *update* is eventually enabled. The results are shown in Table 7.4.

P3: Right price displays or errors If the maps from ESELs to products and from products to price are given to the ESEL system, the right price should be displayed on the linked ESELs for each product, or errors should be reported in case of problems. For this property, it is hard to model in LTL or CTL formula because of its logic. Our solution is to write a *PriceChecker* process in *Circus* to simulate the environment of the ESEL system.

$$\begin{array}{l}
\textbf{section}\ \ \mathit{ESELPriceChecker}\ \textbf{parents}\ \ \mathit{ESELHeader}
\end{array}$$

$$\begin{array}{l}
\textbf{process}\ \ \mathit{PriceChecker} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \ \mathit{State} == [\, \mathit{dummy} : \{0\} \,] \\
\quad \mathit{Init} == [\, (\mathit{State})' \mid \mathit{dummy}' = 0 \,] \\
\quad \mathit{AOnDisplay} \mathrel{\widehat=} \mathit{eid} : \mathit{ESID} \bullet \mathit{ondisplay}.\mathit{eid} \rightarrow \\
\qquad (\interleave\, e : (\mathit{ESID} \setminus (\{\mathit{eid}\})) \,\lbrack\!\lbrack \varnothing \rbrack\!\rbrack \bullet (\mathit{offdisplay}.e \rightarrow \textbf{Skip})) \\
\quad \mathit{AOffDisplay} \mathrel{\widehat=} (\interleave\, e : \mathit{ESID} \,\lbrack\!\lbrack \varnothing \rbrack\!\rbrack \bullet (\mathit{offdisplay}.e \rightarrow \textbf{Skip})) \\
\quad \mathit{ACheckMap} \mathrel{\widehat=} p : \mathit{Price} \,;\, \mathit{eid} : \mathit{ESID} \,;\, \mathit{pid} : \mathit{PID} \bullet \\
\qquad (\mathit{updateallmap}.(\{\mathit{eid} \mapsto \mathit{pid}\}) \rightarrow \mathit{updateallprice}.(\{\mathit{pid} \mapsto p\}) \rightarrow \\
\qquad \mathit{update} \rightarrow \mathit{write}.\mathit{eid}.p \rightarrow (\\
\qquad\qquad (\mathit{read}.\mathit{eid}.p \rightarrow \mathit{AOnDisplay}(\mathit{eid}) \,;\, \mathit{failures}.(\{\}) \rightarrow \textbf{Skip}) \\
\qquad\quad \Box\ (\mathit{read}.\mathit{eid}?x : (x \neq p) \rightarrow \mathit{AOffDisplay}; \\
\qquad\qquad\quad \mathit{failures}.(\{\mathit{pid} \mapsto \{(\mathit{fail}\ \mathit{eid})\}\}) \rightarrow \textbf{Skip}) \\
\qquad\quad) \\
\qquad) \\
\quad \mathit{ACheckNoMap} \mathrel{\widehat=} p : \mathit{Price} \,;\, \mathit{eid} : \mathit{ESID} \,;\, \mathit{pid} : \mathit{PID} \bullet \\
\qquad (\mathit{updateallmap}.(\{\}) \rightarrow \mathit{updateallprice}.(\{\mathit{pid} \mapsto p\}) \rightarrow \mathit{update} \rightarrow \\
\qquad \mathit{AOffDisplay} \,;\, \mathit{failures}.(\{\mathit{pid} \mapsto \{\mathit{NA}\}\}) \rightarrow \textbf{Skip}) \\
\quad \mathit{ACheck} \mathrel{\widehat=} \textbf{var}\ p : \mathit{Price} \,;\, \mathit{eid} : \mathit{ESID} \,;\, \mathit{pid} : \mathit{PID} \bullet \\
\qquad \mathit{ACheckMap}(p, \mathit{eid}, \mathit{pid}) \,\Box\, \mathit{ACheckNoMap}(p, \mathit{eid}, \mathit{pid}) \\
\quad \bullet \big(\mathit{Init}\big) \,;\, \mathit{AOffDisplay} \,;\, (\mu X \bullet (\mathit{ACheck}) \,;\, X) \\
\textbf{end}
\end{array}$$

$$\textbf{channelset}\ \mathit{ESELSystemInterface} == \{\!|\ \mathit{updateallprice}, \mathit{updateprice}, \mathit{updatemap}, \mathit{updateallmap}, \mathit{update}, \mathit{ondisplay}, \mathit{offdisplay}, \mathit{write}, \mathit{read}, \mathit{failures}\ |\!\}$$

This process *PriceChecker* implements a price update and provides an external choice of all possible expected results: successful if write and read values to the display are equal; failed if they are not equal; or *NA* if no ESEL links to the product. Then *PriceChecker* is put in parallel with each *Circus* model to get a new test model.

For the Specification,

$$\textbf{section}\ \mathit{ESELSpecChecker}\ \textbf{parents}\ \mathit{ESELPriceChecker}, \mathit{ESELSpec}$$

$$\textbf{process}\ \mathit{ESELSpecChecker} \mathrel{\widehat=} (\mathit{PriceChecker} \llbracket \mathit{ESELSystemInterface} \rrbracket \mathit{ESELSpec})$$

For the System One,

$$\textbf{section}\ \mathit{ESELSystem1Checker}\ \textbf{parents}\ \mathit{ESELPriceChecker}, \mathit{ESELSystem1}$$

$$\textbf{process}\ \mathit{ESELSystem1Checker} \mathrel{\widehat=} (\mathit{PriceChecker} \llbracket \mathit{ESELSystemInterface} \rrbracket \mathit{ESELSystem1})$$

For the System Two,

$$\textbf{section}\ \mathit{ESELSystem2Checker}\ \textbf{parents}\ \mathit{ESELPriceChecker}, \mathit{ESELSystem2}$$

$$\textbf{process}\ \mathit{ESELSystem2Checker} \mathrel{\widehat=} (\mathit{PriceChecker} \llbracket \mathit{ESELSystemInterface} \rrbracket \mathit{ESELSystem2})$$

Using model checking of ProB, if a test model is deadlock free, then we conclude this property holds in the corresponding *Circus* model. The property check results are illustrated in Table 7.4.

7.3 Steam Boiler Control System

The motivation for this case study is that the steam boiler control specification problem [65] actually has become the standard benchmark for formalisms. The original text of the problem was written by Bauer [96]. Then it was modified and published in the Dagstuhl workshop by Abrial [65] as a competition for formal specifications and development methods [97]. Subsequently, twenty-two solutions, along with the description of the problem, are published in the book [98] while Abrial's own solution is included in the B book [5]. In particular, Woodcock and Cavalcanti [67] proposed a solution using *Circus*. The original specification in *Circus* was published in the report [66] and afterwards Freitas [50] updated it and took it as a test case for the development of CZT. Our study is based on this modified version, and eventually this work corrects and improves the steam boiler solution in *Circus*.

7.3.1 The Problem

The problem aims to design a control system to keep the steam boiler safe because the boiler would be in danger if the water level is too low or too high. The system includes several physical units: the steam boiler, a sensor to measure the quantity of water, four pumps, four pump controllers (one for each pump), a sensor to measure the quantity of steam, an operator, and a message transmission system. In addition, there is a valve in the boiler used in the initialisation stage only to empty the boiler. And all units are characterised by the following constants:

- capacity of the boiler (C in litres),
- water lower and upper limits (M_1 and M_2 in litres),
- maximum and minimum normal working levels (N_1 and N_2 in litres),
- maximum quantity of steam (W in litre/sec),
- capacity of pump (P in litre/sec),
- and maximum increase and decrease gradient of quantity of steam (U_1 and U_2 in litre/sec/sec).

The system has five operation modes:

- *initialisation*: the initial mode,
- *normal*: the standard operating mode (and all physical units operate correctly),
- *degraded*: the degraded mode (some physical units are defective but the water level measuring unit operates correctly),
- *rescue*: the mode in which the system is still working by estimating the water level from the quantity of steam coming out (the water level measuring unit is defective but the steam quantity measuring unit is working),
- *emergencyStop*: the mode that could be reached from any mode if there is a vital unit failure, or the reach of water level to its limits, or a transmission error.

The detailed requirements are described in the paper [65].

7.3.2 *Circus* Solution

The *Circus* solution proposed for the steam boiler control problem consists of four processes: *Timer*, *Analyser*, *Controller* and *Reporter*. The *Timer* process guarantees that each cycle of the control program starts every five seconds. The *Analyser* process takes input from the physical environment, analyses them, and then supplies analysed results to the *Controller* and *Reporter*. The *Controller* is used to manage the switch of different system modes according to analysed results and then report corresponding actions and current mode to the *Reporter*. Finally the *Reporter* gathers information, packages them, sends them to physical units, and notifies the *Analyser* with expected pump states in this cycle. The interaction between these processes is presented in Figure 7.9 from the diagram [66, Figure 1].

After introducing our model checking solution in this thesis to the original *Circus* model, we have made some changes to the model as described in Section 7.3.3 and a number of corrections as stated in Section 7.3.4. The corrected and updated model is shown in Appendix I.1.

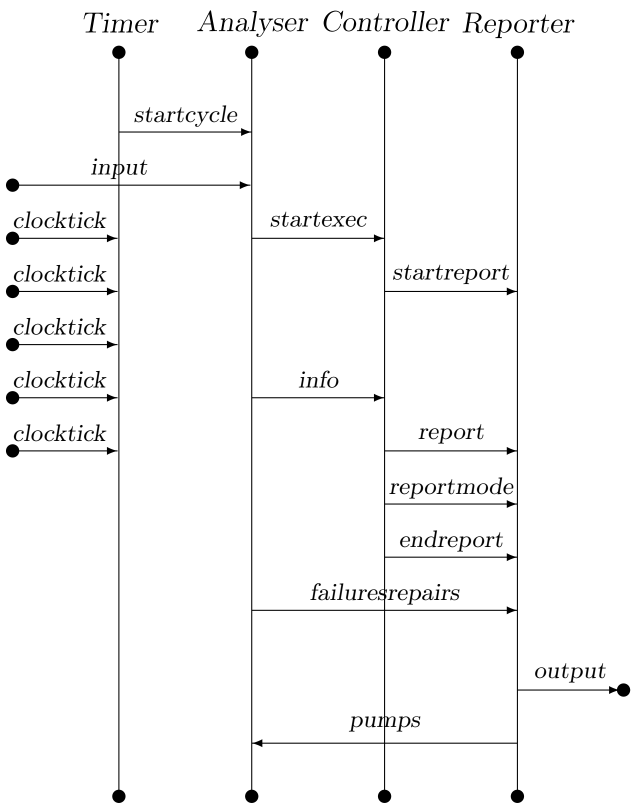

Figure 7.9: Message Sequence Chart [66]

7.3.3 Practical Considerations

7.3.3.1 Input and Output Channels

The *input* channel is originally of the schema type *InputMsg* which encapsulates all input messages (*signals*, *pumpState*, *pumpCtrState*, *q*, *v*, *failureacks*, and *repairs*) from the physical environment. When animating the specification using ProB, it takes a very long time to calculate all possible values and load them on ProB. It is the same case as the *output* channel. To mitigate it, we break the input messages into seven small messages and each for one variable in *InputMsg*. Consequently, all input messages are considered to arrive in serial instead of simultaneously. Eventually, the *input* channel declaration becomes seven channel declarations

$$\begin{array}{l}
\textbf{channel}\ \mathit{input}1 : (\mathbf{P}\ \mathit{InputSignal}) \\
\textbf{channel}\ \mathit{input}2 : (\mathit{PumpIndex} \rightarrow \mathit{InputPState}) \\
\textbf{channel}\ \mathit{input}3 : (\mathit{PumpIndex} \rightarrow \mathit{InputPCState}) \\
\textbf{channel}\ \mathit{input}4 : (\mathit{NUMS}) \\
\textbf{channel}\ \mathit{input}5 : (\mathit{NUMS}) \\
\textbf{channel}\ \mathit{input}6 : (\mathbf{P}\ \mathit{UnitFailure}) \\
\textbf{channel}\ \mathit{input}7 : (\mathbf{P}\ \mathit{UnitFailure})
\end{array}$$

and the *input* communication in *Analyser* becomes seven channel events.

$$\begin{array}{l}
\mathit{startcycle} \rightarrow \mathit{input}1?\mathit{signals} \rightarrow \mathit{input}2?\mathit{pumpState} \rightarrow \mathit{input}3?\mathit{pumpCtrState} \rightarrow \\
\mathit{input}4?q \rightarrow \mathit{input}5?v \rightarrow \mathit{input}6?\mathit{failureacks} \rightarrow \mathit{input}7?\mathit{repairs} \rightarrow \cdots
\end{array}$$

The *output* channel is analogous to the *input* channel.

7.3.3.2 Loose Constants

The specification has nine constants: C, P, U_1, U_2, W, M_1, M_2, N_1, and N_2. They are all of type $\mathbb{N}$ and loosely related. The loose constants really complicate model checking. Together with the limitation—only one instance of constants can be model-checked at once—stated in Section 6.5.2, we choose one configuration of all constants for model checking. Additionally, we change the type $\mathbb{N}$, that is for most of variables, to the type $NUMS$ ($NUMS = 0..MAX\_NUM$, where MAX_NUM is a constant as well), consequently, from infinite type to finite.

7.3.3.3 Specific Initialisation

Not all initial values of state components matters, and the state variables in *InputMsg* are among them. This initialisation is completely correct. However, practically it makes model checking harder because the state space becomes much larger, which is very similar to loose constants. Thus, we add specific initialisation to these variables for model checking purpose. Regarding *InputMsg*, we add a predicate ($\theta\, InputMsg' = \cdots$) in *InitAnalyserState* to specify its initial state.

The *Timer* process and the *Reporter* process are similar. We add a *InitTimer* and a *InitReporter* in *Timer* and *Reporter* respectively.

7.3.4 Errors Found and Corrections Made

Taking advantage of rich syntax and expressiveness in *Circus*, the specification of the steam boiler is well designed and structured. However, at the time of writing the report [23], the *Circus* language itself is in the very early stage of development, and no parser and type checker are available. Afterwards, its syntax, semantics and tools are continuously developed. As a result, there are some inconsistent and incorrect constructs found by current parser and typechecker in CZT. Additionally, after introducing animation and model checking into *Circus*, more errors are found and corrected.

7.3.4.1 During Parsing and Type-Checking

- The *input* channel is not declared.
- The name of the channel *failures* conflicts with that of the *failures* variable in the *Failures* schema. Hence, the channel is renamed to *cfailures*.
- To eliminate the confusion of pa_1 and pa_2 in $Pump0$ and *PumpCtrSystem* schemas, the variables in *PumpCtrSystem* are renamed to pta_1 and pta_2 that mean the lower-bound and upper-bound total adjusted pump volumes.
- There is no state schema in the *Reporter* process. So the schema *ReporterState*, having *OutputMsg* as state component, is added in *Reporter*.
- No definition of the *FailuresRepairs* schema is found in *Reporter* and thus it is added as well to update output messages according to (*noacks*, *repairs*) from the *Analyser* process.

7.3.4.2 During Animation

Animation plays a significant role in writing, understanding, and correcting the specification. Using ProB, we can animate the specification manually or automatically to check if it works as expected. It is the first step, after parsing and type-checking, in verifying the specification. We use two methods to explore the system via animation: manual animation by selecting tests to enter each operation mode of the system, and automatic random animation for a comparatively long time. However, animation cannot guarantee the correctness of the system. Thus we shall not rely heavily on it. Finally, the errors found during animation are listed below.

- The time operation in *Timer*, ($time := time + 1 \bmod cycletime$), should be ($time := (time+1) \bmod cycletime$) because the modulo operator (mod) has a higher precedence than the addition (+) operator.
- The computation of vc_2 in *VUpperBound* should be + instead of −.
- The check of $x?$ in the *Expect* schema should compare to calculated values (c_1 and c_2).
- The quantifiers in *SetPumpCtr* is corrected to

$$\begin{aligned} &SetPumpCtr == \forall\, i : PumpIndex \bullet \exists\, PumpCtr\, ; PumpCtr\,'; \\ &\qquad pst?, exppst : PState\, ; pcst?, exppcst : PCState \bullet (\cdots) \end{aligned}$$

- The comparison of qa_2 in *DangerZone* should be with N_2, instead of N_1.
- The expected pump state *expectedp* and pump control state *expectedpc* are not updated. Therefore, a schema *CalcExpectedPumpState* is added to set *expectedp* and *expectedpc* according to the pump state which is sent by the control system to the physical unites.
- The state *valve* is assessed in *QLowerBoundValveOpen* and *QLowerBoundValveClosed*, but it is not assigned and updated. Thus a free type *VStateAct* and a schema *SetValveState* are added to update *valve*, and a state component *valveSt* is appended in *Reporter*. Furthermore, the *pumps* channel is extended to accommodate *valveSt* information for *Reporter* to update *Analyser* with valve information.
- The *failures* is calculated based on the comparison of input states with expected states in each cycle, and the *noacks* is updated in *AcceptFailureAcks* and *AcceptRepairs*. However, *noacks* is not assigned and *failures* is not repaired in the specification. Eventually, we refine the requirements of failures as below.
  - The life cycle of a failure consists of several stages. To begin with, a new failure is detected if the input state of an equipment is not the same as its expected state, and at the same time the failure is added in *noacks*. Then the failure is sent to the physical unit by the control program. Afterwards, it might be acknowledged or repaired by the physical unit, and subsequently it is taken out of *noacks* or *failures* respectively.
  - A failed equipment will turn back to its normal state only after it is acknowledged and repaired.

  According to the requirements, the schema *UpdateFailuresAck* is added to assign and update *noacks*, and the schema *CheckAndAdjustPump* is updated to comply the requirement that if a pump fails, its state will not be changed in this schema. Furthermore, a schema *RepairEquipments*, that includes *RepairPumps*, *RepairVSensor*, and *RepairQSensor*, is added accordingly to repair equipments.
- In the *EmergencyStopCond*, the predicate ($\neg$*RepairsExpected* or $\neg$*FailuresExpected*) is used to check if the input *repairs* and *failureacks* are expected. However, since this schema is evaluated in the *InfoService* action which is in the later stage of *Analyser*, the *failures* and *noacks* that are assessed in *RepairsExpected* and *FailuresExpected* respectively are not the original values in the beginning of cycle. Instead, they

are modified in the *Analyse* schema. Therefore, it is not correct to check *repairs* and *failureacks* against updated *failures* and *noacks*. To fix it, an additional state component named *emergencyCond* is added, and it is assigned in the early stage of cycle (*HandleRepair*) and used in *EmergencyStopCond* later.

7.3.4.3 During Deadlock Checking

Though problems have been found during parsing, type-checking, and animation, there are some hidden issues that might not be found easily by them. Using deadlock checking, we found additional problems in the specification.

- One deadlock occurs and the counterexample shows the precondition of *Analyse* does not hold. Actually, it diverges instead of deadlocks according to the semantics of the schema expression in *Circus*—if the precondition does not hold, it diverges. However, we define DIV as $div \rightarrow STOP$ and use the deadlock checking of ProB to find divergence of the schema expression. Finally we found the cause of the problem. It is because the pump's state *pst* in *CheckAndAdjustPump* is unexpectedly updated to *popen* even its current state is *pfailed*. That is against our requirement that a failed equipment is only back to normal state by a repair. Consequently, the predicate $noacks \subseteq failures$ in *Failures* is not satisfied, and the precondition of *Analyse* does not hold. Our solution is to correct the predicate in *CheckAndAdjustPump*.

- A deadlock occurs when *Controller* enters *emergencyStop* mode and it is waiting for synchronisation of *AdjustLevel* from *Analyser*. However, at that time, *Reporter* has been in *TidyUp* and is waiting for synchronisation of *endreporter* from *Controller*. We modified *AdjustLevel* in *ControllerCycle* to make it only adjust levels when it is not in the *emergencyStop* mode.

$$\big((mode \notin Nonemergency) \mathbin{\&} AdjustLevel \mathbin{\Box} (mode = emergencyStop) \mathbin{\&} Skip\big)$$

- A deadlock occurs when the precondition of *Analyser_Analyse* does not hold. Using "Debug operation PRE" function of ProB, we found calculated *qc_2* in *QUpperBound* is less than 0, and it is against its declaration—it must be natural numbers. Therefore, we modified the predicate of *QUpperBound* to

$$\begin{array}{l} QUpperBound == [\,CValues\,;\,QSensor\,;\,VSensor\,;\,PumpCtrSystem \mid \\ \quad qc\_2 = max\{0, min\{C, qa\_2 - 5 * va\_1 + 12 * U\_2 + 5 * pta\_2\}\}\,] \end{array}$$

7.3.4.4 During Property Checking

Apart from the deadlock checking—safety property, other properties can be derived from the requirements and expressed by LTL or CTL. Then they are verified on ProB by temporal logic checking. We have attempted to check properties below.

P1 The signal *openValve* and *closeValve* are mutually exclusive and shall not be sent to the physical units at the same time.

We use the CTL formula below.

$$AG(not(\{openValve \in Reporter\_signals \land closeValve \in Reporter\_signals\}))$$

The formula is checked to be *false* and the counterexample shows the problem. It is because the *OutputMsg* in *Reporter* is kept to the next cycle and it should be cleared for each cycle. Therefore, we correct the main action of *Reporter* to

$$\bullet\ \mu X \bullet startreport \rightarrow \big(InitReporter\big)\,;\,ReportService\,;\,X$$

Table 7.5: Deadlock Checking Performance

No	NP [1]	SoC [2]	C	M_1	M_2	N_1	N_2	P	W	U_1	U_2	Time (min)	Mem (GB)	Checked
1	1	2	12	2	10	4	8	1	2	1	1	1,158	180	32%
2	1	1	12	2	10	4	8	1	2	1	1	2,413	147	38%

[1] Number of Pumps
[2] Seconds in one cycle [sec]

7.3.5 Performance and Results

The control system has rich states and its state space is too large to be solved by ProB. Our attempt to model-check this system on ProB on a server having 128GB RAM and 32-core CPU, and running Ubuntu 14.04.4, does not achieve the result after five days. In addition, we tried to simplify the system by reducing the number of pumps from four to one, changed to smaller constants and cycle time, and tested on another server with high specification (512GB RAM). Nevertheless, the system is still not model-checked completely and the intermediate result is displayed in Table 7.5. Note that 2,186,002 states and 8,272,654 transitions are checked in the first row, and 3,855,002 states and 12,339,468 transitions are checked in the second row.

7.3.6 Experiences and Lessons Learned

- For schema expressions, if designed without extra cautions, it is highly possible that the system will diverge due to the preconditions that is not satisfied. However the divergence is not always what we want. Therefore, it is helpful to consider all cases in their preconditions and make them hold always.

- Compared to a schema expression that specifies all state components in its predicate, it might be simpler in design to decompose the schema expression into multiple schema expressions and make each one update a part of the state components. Finally the system can benefit from this simplification in design as well as verification.

- For state-rich reactive systems, their model checking is more complicated than that of state-based or behaviour-oriented systems. To address the state space exploration problem, it is essential for model checkers to support multiprocessors and distributed system, as well as some optimisations to reduce the state space.

7.4 Summary

In this chapter, three case studies have been presented.

In the reactive buffer case, we demonstrate the stepwise application of our link rules to *Circus* models. Both the buffer specification and the buffer implementation are linked to two corresponding $CSP \parallel_B Z$ models. Using ProB, we have checked deadlock free and livelock free for both models with a number of constants configurations as shown in Table 7.1 and Table 7.2. In addition, we also have checked the implementation is a correct refinement of the specification in terms of traces and failures models as shown in Table 7.3 and the reasoning of data independence in Section 7.1.3.2.

Then the development of a system from requirements, to specification, then to more specific systems by *Circus* is illustrated in the ESEL case. That results in three *Circus* models: the ESEL Specification, the System One, and the System Two. All of them are translated to $CSP \parallel_B Z$ and using ProB we have checked they are deadlock free and livelock free as shown in Table 7.4 when MAX_ESEL is equal to 2 and 3. Furthermore, both the System One and the System Two are checked to be traces and failures refinements of the Specification when both MAX_ESEL and $MAX\_GATEWAY$ are equal to 2. In the end, three application related properties have been checked as well. This is shown in the table too.

For the steam boiler case, though it is not completely checked due to its large state space, the case study still shows our model checking approach is beneficial to the system in design. With animation and model checking in ProB, many errors are identified and corrected, and finally the design has been improved dramatically. With animation, it also helps users to understand how the steam boiler system works.

Among three cases, the bounded buffer is popular as it has been studied in [64] for refinement strategies and in the *map* solution [55] for its model checking solution. However, no performance metrics are presented. In addition, since the ESEL case is first presented in this thesis and for the steam boiler solution in *Circus*, it is the first attempt to verify it, comparison of performance with other model checking solutions is also not possible.

In sum, the three case studies presented in this chapter illustrate the usability of our solution as well as the limitations. From another angle, they indicate the direction of future improvement.

Chapter 8

Conclusion

In this chapter, a summary of the thesis is given first. Then we summarise contributions of our work as well as limitations of our solution. In the end, a discussion of future work is presented.

8.1 Thesis Summary

Computer-based systems are becoming more and more complex, from both individual systems and systems of systems due to increasing interaction. To address the growing complexity issue, interest in the use of formal methods has increased rapidly because of its main advantage, elimination of ambiguity. It is extremely important especially for safety critical systems that require a high level of assurance.

Circus is a state-rich formal language designed to address this issue. It is a combination of Z, CSP, refinement calculus and Dijkstra's guarded commands. Thanks to its rich notations, it can model not only very abstract specifications, but also very specific implementations of systems. Most importantly, its calculational-style refinement distinguishes itself from other integrated formal methods. In addition, *Circus* has its semantics defined on UTP, which makes it extendable to support heterogeneous systems. Due to these features of *Circus*, it is very difficult to develop tools to support its formal verification from both theorem proving and model checking. There are some existing tools developed as discussed in Chapter 1. Especially for model checking, we discuss it in depth in Chapter 3. In the chapter, a survey of current verification tools is provided and then we analyse their advantages and disadvantages. The review of current tools motivated our approach to model-check state-rich formalisms (specifically *Circus*) as presented in this thesis. Our approach is to link *Circus* to $CSP \parallel B$ formally and then use ProB to model-check $CSP \parallel B$.

Since our approach is to establish the link from *Circus* to $CSP \parallel B$, and soundness of the link is based on UTP, a basic knowledge of UTP, *Circus*, and $CSP \parallel B$ is given in Chapter 2. For UTP, the theories about alphabetised relational calculus, refinement, designs, reactive processes, and CSP processes are introduced. These are the theoretical aspect of *Circus* denotational semantics. Particularly, we give a detailed description of *Circus* syntax because our later link definition and soundness highly rely on the understanding of its rich notations. Furthermore, in order to get a quick understanding of these notations and their usage, an example of a *Circus* model, the ESEL specification, is presented in the chapter. Actually, the example is a part of the ESEL case study in Chapter 7. By moving it to this early chapter, we intend to give readers a first impression of a *Circus* model. In the end, the introduction of $CSP \parallel B$ mainly focuses on the understanding of its model in terms of CSP. We define a new notation $\,_{C}\|_{B}$ to denote it and a number of laws to facilitate the semantic reasoning of $CSP \parallel B$ models.

The link of our solution is formally defined in Chapter 4. A function Υ is defined to denote the link from *Circus* to $CSP \parallel B$. In order to support separation of the state part and the behaviour part from *Circus*, Υ is decomposed into three sub-functions:

- R_{wrt} to rewrite original *Circus* models to the written models in which all interactions between the state part and the behaviour part are reduced to schema expressions as action. After that, the written models are ready to be separated.
- Ω to link the state part of written models to B. Since the Z dialect in *Circus* is ISO Standard Z and the final language is B, in order to reuse the translator in ProB to translate Z in ZRM to B, Ω is also further decomposed into three sub-functions:
  - Ω_1 is used to extract the state part from a written model, and eventually form a Z model in ISO Standard Z. And this new model will be parsed and typechecked before moving to the next sub-function.
  - Ω_2 is defined to syntactically transform the Z model in ISO Standard Z to the Z model in ZRM.
  - Ω_3 is for the translation of the Z model in ZRM to B. In our solution, we rely on the translator in ProB for this function.
- Φ to link the behaviour part of the written models to CSP.

Then in the rest of the chapter, for each sub-function we present individual rules for each construct in *Circus* from global definitions, to processes definitions, and finally to actions. Particularly, the Φ rule for *Circus* expressions, predicates, and operators has its map listed in Appendix D. Furthermore, Appendix E summarises the link rules from each construct in *Circus* to $CSP \parallel_B Z$ by the combination of these sub-functions. These link rules are very useful in the proof of soundness afterwards.

After presentation of the link definition, we reason about it in Chapter 5. Our strategy is to present the denotational semantics of a *Circus* construct, use link rules in Appendix E to link it to $CSP \parallel_B Z$, then give the semantics to the result $CSP \parallel_B Z$ construct, and finally compare the semantics of both constructs. If they have the same semantics, we conclude the link rules used for this construct in *Circus* are sound. If the link rules for all constructs defined in our link are sound, then our link is sound. In particular, the soundness of variable block is based on Theorem C.1.28 which is defined and proved in Appendix C.

In order to make the translation automatic, we developed a translator, *Circus2ZCSP*, based on CZT. The overall translation procedure and a number of key algorithms, along with practical considerations, are presented in Chapter 6.

Finally, three case studies are presented in Chapter 7 to demonstrate the usability of our solution. Three case studies are illustrated in order of ascending complexity. In the reactive buffer case, we demonstrate the stepwise application of our link rules to *Circus* models. Then the development of a system from requirements, to specification, then to more specific systems by *Circus* is illustrated in the ESEL case. Finally, for the real industrial steam boiler case, though it is not completely checked due to its large state space, the case study still shows our model checking approach is beneficial to the system in design. With animation and model checking in ProB, many errors are identified and corrected, and finally the design has been improved dramatically. Three case studies presented in this chapter also show current limitations of our solution.

8.2 Contributions

We propose a solution to model-check *Circus* by linking it to $CSP \parallel B$. Our contributions of this work are summarized as follows.

1) We define a formal link from *Circus* to $CSP \parallel B$ and its soundness is based on the UTP semantics. Therefore, we also give the UTP semantics to $CSP \parallel B$.

2) We developed a translator to link *Circus* to $CSP \parallel B$ automatically. The translator supports nearly all constructs that are defined in our link.

3) Three case studies with different complexity have demonstrated the usability of our approach.

4) Our lessons learned and experience we got during this work would be helpful to develop similar solutions for *Circus*.

Furthermore, the modifications that we made to ProB, no matter whether they are merged in ProB releases or not, provide improvements to ProB or valuable insights into the possible solutions and implementations for ProB to better support $CSP \parallel Z$ in terms of $CSP \parallel B$. These modifications are listed as below.

1) Unification of sequence types between CSP and Z is supported as stated in Section 7.1.3.4. It has been adopted by the ProB team and merged into ProB releases.

2) Support of free types in CSP is added to ProB. In addition, unification of free types between CSP and Z is supported as well. This modification is described in Section D.2.1.1. It implements our solution only and has not been adopted by the ProB team yet.

3) Support of schema types in CSP is added to ProB. In addition, unification of schema types between CSP and Z is supported as well. This modification is described in Section D.2.1.2. It implements our solution only and has not been adopted by the ProB team yet.

4) The well-definedness of the modulo operator in Z, as described in Section 7.1.3.5, has been corrected in ProB and this change has been merged into current ProB release.

5) Refinement checking in ProB has been updated to support $CSP \parallel B$ as stated in Section 7.1.3.6. This change has not been adopted by the ProB team yet.

8.3 Limitations

Throughout the thesis, we mention the limitations of our solution in different chapters. In this section, we summarise our solution's limitations as follows.

8.3.1 Mutual Recursive Freetype

It is not supported because it is defined only in ISO Standard Z and not in ZRM. See Ω_2 Rule 3.

8.3.2 Operator Template and User Defined Operators

It is not supported by the translator of Z to B in ProB.

8.3.3 Parallel Composition and Interleaving of actions

As stated in Φ Rule 32, general parallel composition and interleaving of actions are not supported in our link. But two special cases are supported. One is "Disjoint Variables in Scope" in Φ Rule 30 and another is "Disjoint Variables in Updating" in Φ Rule 31. Additionally, iterated parallel composition and interleaving of actions are not supported.

8.3.4 External Choice of Actions

Our link restricts actions in an external choice to be prefixed actions given in Definition B.3.1. Particularly, recursion has an additional restriction to be an action in the external choice in addition to prefixed actions. We require initial events of the action in a recursion do not evaluate state variables. Otherwise, it will cause problems as stated in R_{wrt} Rule 34. Though it is a restriction to be prefixed actions in external choice, it would not reduce expressive power of *Circus*. We can rewrite the external choice to avoid state changes before initial events of actions. One possible way is to add guards before each action to make it an external choice of guarded actions and at the same time only one guard is true.

8.3.5 External Choice of Processes

Our link requires both processes in an external choice of processes are prefixed processes defined in Definition B.4.1 as described in R_{wrt} Rule 22. The reason of this limitation is because the limitation of external choice of actions above because the external choice of processes is semantically equal to the external choice of their main actions.

8.3.6 Schemas as Predicates

For schemas as predicates in the behavioural part of *Circus*, they have been translated only in two constructs: predicates as boolean expressions in channel output expressions and predicates as conditions in guarded actions.

For a schema as predicate, when it is rewritten to a predicate, our translator takes all predicates from the declaration part and the predicate part, and combines them together according to schema operators. It may result in duplicate predicates from the declaration part. It will not cause problems but make the final CSP model larger if there are many schema references in the schema as predicate.

Both limitations are due to implementation of the translator.

8.3.7 Recursion in Variable Block

The recursion cannot be an action of variable block according to Theorem C.1.28.

8.3.8 Partial Looseness Loss

For axiomatic definitions, if they are used in CSP, they have to be instantiated to become concrete and specific in advance. Our solution cannot cope with loose constants in the behavioural part of *Circus*. The behavioural model checker FDR does not support loose constants too.

Furthermore, looseness may come from uninitialised state space which also increases the complexity of model checking. Our solution has limited support of this looseness if the state space is relatively small.

8.3.9 Partial Abstraction Loss

Since CSP does not support abstract types such as $\mathbb{N}$, the translation of an abstract *Circus* specification into a concrete CSP specification will result in information loss. Our solution uses a B machine to represent the state part, and consequently preserve the abstraction. However, if these abstract types are used in the behavioural part, our solution has the same issue as the translation from *Circus* to CSP directly. Eventually, our solution may have partial loss of abstraction.

8.3.10 Model Checking Performance

Our solution highly relies on ProB to model-check the final $CSP \parallel B$ model. Since ProB cannot support multithreading or multiprocessing, and only one process per ProB instance is allowed, the size of state space that can be model-checked is limited. Our case studies have shown this issue.

8.4 Future Work

The limitations above indicate the direction of our future work.

8.4.1 Automatic Checking of Multiple Models

Section 7.1.3.3 describes the feature which enables ProB to check multiple models at once automatically.

8.4.2 Operator Template and User Defined Operators

Operator templates that are frequently used in *Circus* have only syntactic significance. They notify readers or their parser to treat all occurrences of the words in the templates as specific operators, such as function, relation, or generic. They also give particular information about prefix, postfix, infix, or nofix of the operators defined. To support operator template, we can transform syntactically in our rewrite rules or leave them in the later stage in ProB and then extend ProB to support them.

8.4.3 Parallel Composition and Interleaving of Actions

It is not too difficult to define the link rule for general parallel composition and interleaving of actions. The solution is to declare two copies of current variables in scope, make each action only update its own copy, and then finally merge updates of variables, that correspond to variables in each action's name set, in temporary copies. However, the difficulty arises from its implementation in the translator. The main problem is how to update all variables in temporary copies instead of original variables in both actions. It is still possible by rewriting both actions but left as future work.

8.4.4 External Choice of Actions

Similar to parallel composition and interleaving of actions, for external choice of actions, in order to preserve its semantics that state change will not resolve it, one possible solution is to declare two copies of variables in scope of both actions, and then discard another one only after its resolution by events or termination. This can be implemented in ProB by extending its kernel to support this new external choice. The lazy evaluation and backtracking of Prolog would be helpful to implement it.

8.4.5 Performance Improvement

The most important and also hardest aspect is to improve the performance of model checking. On the one hand, as an extension of our work in this thesis, we could collaborate with the ProB team together to introduce the optimisations, such as symmetry reduction [99], partial order reduction, and partial guard evaluation [100], into $CSP \parallel B$. To some extent, for some specific cases, it might be useful to shrink the state space. But it might be improbable to improve performance dramatically for industrial scale cases. However, if ProB could support multithreading and multiprocessing in the future, it would be very useful and helpful to our solution.

On the other hand, we may look at other model checkers in addition to ProB. Since the main objectives of our solution are to maintain abstraction and looseness, the solution of translating *Circus* specifications into CSP is not what we expect from this aspect. For state-rich systems, we could assume their models in *Circus* are largely composed of the state part and have simple behavioural specification. From this point of view, a possible solution is to translate *Circus* to TLA+ [101] and then use TLC [102], a model checker for TLA+, to model-check the resultant TLA+ models. We still can use the similar link defined in this thesis. But instead of the translation of Z to B by Ω_3, we need a translation from Z to TLA+. Additionally, a translation from CSP to the temporal logic formula in TLA+ is another extra work to replace our Φ function.

Appendix A

Circus Syntax

$$
\begin{array}{lll}
\mathrm{Program} & ::= & \mathrm{CircusPar}^{*} \\
\mathrm{CircusPar} & ::= & \mathrm{Par} \mid \mathbf{channel}\ \mathrm{CDecl} \mid \mathbf{chanset}\ \mathrm{N} == \mathrm{CSExp} \mid \mathrm{ProcDecl} \\
\mathrm{CDecl} & ::= & \mathrm{SimpleCDecl} \mid \mathrm{SimpleCDecl;CDecl} \\
\mathrm{SimpleCDecl} & ::= & \mathrm{N}^{+} \mid \mathrm{N}^{+} : \mathrm{Exp} \mid [\mathrm{N}^{+}]\mathrm{N}^{+} : \mathrm{Exp} \mid \mathrm{SchemaExp} \\
\mathrm{CSExp} & ::= & \{\!|\ |\!\} \mid \{\!|\ \mathrm{N}^{+}\ |\!\} \mid \mathrm{N} \mid \mathrm{CSExp} \cup \mathrm{CSExp} \mid \mathrm{CSExp} \cap \mathrm{CSExp} \\
 & \mid & \mathrm{CSExp} \setminus \mathrm{CSExp} \\
\mathrm{ProcDecl} & ::= & \mathbf{process}\ \mathrm{N} \mathrel{\widehat=} \mathrm{ProcDef} \mid \mathbf{process}\ [\mathrm{N}^{+}]\mathrm{N} \mathrel{\widehat=} \mathrm{ProcDef} \\
\mathrm{ProcDef} & ::= & \mathrm{Decl} \bullet \mathrm{ProcDef} \mid \mathrm{Decl} \odot \mathrm{ProcDef} \mid \mathrm{Proc} \\
\mathrm{Proc} & ::= & \mathbf{begin}\ \mathrm{PPar}^{*}\ \mathbf{state}\ \mathrm{N} \mathrel{\widehat=} \mathrm{SchemaExp}\ \mathrm{PPar}^{*} \bullet \mathrm{Action}\ \mathbf{end} \\
 & \mid & \mathrm{Proc} ; \mathrm{Proc} \mid \mathrm{Proc} \Box \mathrm{Proc} \mid \mathrm{Proc} \sqcap \mathrm{Proc} \mid \mathrm{Proc}\ \llbracket\ \mathrm{CSExp}\ \rrbracket\ \mathrm{Proc} \\
 & \mid & \mathrm{Proc} \interleave \mathrm{Proc} \mid \mathrm{Proc} \setminus \mathrm{CSExp} \mid \mathrm{Proc}[\mathrm{N}^{+} := \mathrm{N}^{+}] \mid \mathrm{N}[\mathrm{Exp}^{+}] \mid \mathrm{N} \\
 & \mid & (\mathrm{Decl} \bullet \mathrm{ProcDef})\,\big(\mathrm{Exp}^{+}\big) \mid \mathrm{N}\,\big(\mathrm{Exp}^{+}\big) \mid (\mathrm{Decl} \odot \mathrm{ProcDef})\,\lfloor \mathrm{Exp}^{+} \rfloor \\
 & \mid & \mathrm{N}\lfloor \mathrm{Exp}^{+} \rfloor \mid \mathop{;} \mathrm{Decl} \bullet \mathrm{Proc} \mid \Box\, \mathrm{Decl} \bullet \mathrm{Proc} \mid \bigsqcap \mathrm{Decl} \bullet \mathrm{Proc} \\
 & \mid & \llbracket \mathrm{CSExp} \rrbracket\ \mathrm{Decl} \bullet \mathrm{Proc} \mid \big|\big|\big|\ \mathrm{Decl} \bullet \mathrm{Proc} \\
\mathrm{PPar} & ::= & \mathrm{Par} \mid \mathrm{N} \mathrel{\widehat=} \mathrm{ParAction} \mid \mathbf{nameset}\ \mathrm{N} == \mathrm{NSExp} \\
\mathrm{ParAction} & ::= & \mathrm{Action} \mid \mathrm{Decl} \bullet \mathrm{ParAction} \\
\mathrm{Action} & ::= & \big(\mathrm{SchemaExp}\big) \mid \mathrm{Command} \mid \mathrm{N} \mid \mathrm{CSPAction} \mid \mathrm{Action}[\mathrm{N}^{+} := \mathrm{Exp}^{+}] \\
\mathrm{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathrm{Comm} \rightarrow \mathrm{Action} \mid \mathrm{Pred}\ \&\ \mathrm{Action} \\
 & \mid & \mathrm{Action} ; \mathrm{Action} \mid \mathrm{Action} \Box \mathrm{Action} \mid \mathrm{Action} \sqcap \mathrm{Action} \\
 & \mid & \mathrm{Action}\ \llbracket\ \mathrm{NSExp} \mid \mathrm{CSExp} \mid \mathrm{NSExp}\ \rrbracket\ \mathrm{Action} \\
 & \mid & \mathrm{Action}\ \|[\ \mathrm{NSExp} \mid \mathrm{NSExp}\ ]\|\ \mathrm{Action} \\
 & \mid & \mathrm{Action} \setminus \mathrm{CSExp} \mid \mathrm{ParAction}(\mathrm{Exp}^{+}) \mid \mu\ \mathrm{N} \bullet \mathrm{Action} \\
 & \mid & \mathop{;} \mathrm{Decl} \bullet \mathrm{Action} \mid \Box\, \mathrm{Decl} \bullet \mathrm{Action} \mid \bigsqcap \mathrm{Decl} \bullet \mathrm{Action} \\
 & \mid & \llbracket \mathrm{CSExp} \rrbracket\ \mathrm{Decl} \bullet \llbracket \mathrm{NSExp} \rrbracket\ \mathrm{Action} \mid \big|\big|\big|\ \mathrm{Decl} \bullet \|[\mathrm{NSExp}\ ]\|\ \mathrm{Action} \\
\mathrm{Comm} & ::= & \mathrm{N}\ \mathrm{CParameter}^{*} \mid \mathrm{N}\ [\mathrm{Exp}^{+}]\ \mathrm{CParameter}^{*} \\
\mathrm{CParameter} & ::= & ?\mathrm{N} \mid ?\mathrm{N} : \mathrm{Pred} \mid !\mathrm{Exp} \mid .\mathrm{Exp} \\
\mathrm{Command} & ::= & \mathrm{N}^{+} := \mathrm{Exp}^{+} \mid \mathbf{if}\ \mathrm{GActions}\ \mathbf{fi} \mid \mathbf{var}\ \mathrm{Decl} \bullet \mathrm{Action} \\
 & \mid & \mathrm{N}^{+} : [\mathrm{Pred}, \mathrm{Pred}] \mid \{\mathrm{Pred}\} \mid [\mathrm{Pred}] \\
 & \mid & \mathbf{val}\ \mathrm{Decl} \bullet \mathrm{Action} \mid \mathbf{res}\ \mathrm{Decl} \bullet \mathrm{Action} \mid \mathbf{vres}\ \mathrm{Decl} \bullet \mathrm{Action} \\
\mathrm{GActions} & ::= & \mathrm{Pred} \rightarrow \mathrm{Action} \mid \mathrm{Pred} \rightarrow \mathrm{Action} \Box \mathrm{GActions}
\end{array}
$$

Figure A.1: Syntax of Circus

Appendix B

Definitions

Some definitions that are used in this thesis are listed in this appendix.

B.1 Memory Model in CSP

In *Circus*, local variables in an action, such as variable block, can be updated and retrieved very easily because it is a mixture of Z and CSP. However, when the action is transformed to the counterpart in CSP, the variables in CSP process cannot be updated directly. We use the memory model [54, 103] to save and retrieve the value of local variables.

B.1.1 *MemCell* Process

B.1.1.1 Definition

Definition B.1.1 (*MemCell* Process). *A MemCell process defined below is the mechanism in CSP to store the value of a local variable.*

$$\begin{aligned}
MemCell_i =& \quad set_i?x \rightarrow MCell_i(x)\\
MCell_i(x) =& \quad set_i?y \rightarrow MCell_i(y)\\
& \Box\; get_i!x \rightarrow MCell_i(x)\\
& \Box\; end \rightarrow SKIP
\end{aligned}$$

For each local variable, it shall have a MemCell process. Therefore, the process is distinguished by a number i which is a unique number for each variable. The MemCell process is initialized by set_i at first, and after that it will continuously provide update and retrieve of the variable by set_i and get_i channels respectively. Additionally, it is capable of terminating successfully through end event.

B.1.2 F_{Var} Function

Definition B.1.2 (F_{Var} function). *The $F_{Var}(P, v)$ function has a CSP process P and a set of local variables v in this process as inputs, and outputs a CSP process by making every access to each local variable l in v by $get_i?l$ and every update to local variable l by $set_i!l$. For example,*

$$\begin{aligned}
&F_{Var}\,(c?x!y!z \rightarrow P, \{x, y, z\})\\
&\qquad = get_i?y \rightarrow get_j?z \rightarrow c?x!y!z \rightarrow set_k!x \rightarrow F_{Var}\,(P, \{x, y, z\})\\
&F_{Var}\,(c?x!y!z \rightarrow P, \{y, z\})\\
&\qquad = get_i?y \rightarrow get_j?z \rightarrow c?x!y!z \rightarrow F_{Var}\,(P, \{y, z\})\\
&F_{Var}\,(P, \{\}) = P\\
&F_{Var}\,(P, v) = P \qquad if\ lclV\,(P) \cap v = \varnothing
\end{aligned}$$

provided *x, y and z are local variables.*

B.1.2.1 F_{VarPre} and $F_{VarPost}$ Function

Definition B.1.3 (F_{VarPre} and $F_{VarPost}$ function). *The $F_{VarPre}(P, v)$ function gives the prefixing of P in $F_{Var}(P, v)$ (only get_i events in this prefixing) while the $F_{VarPost}$ function gives the remaining of $F_{Var}(P, v)$. $F_{Var}(P, v)$ is composed of $F_{VarPre}(P, v)$ and $F_{VarPost}(P, v)$. For example,*

$$\begin{aligned}
&F_{VarPre}(c?x!y!z \to P, \{x, y, z\}) = get_i?y \to get_j?z\\
&F_{VarPost}(c?x!y!z \to P, \{x, y, z\}) = c?x!y!z \to set_k!x \to F_{Var}(P, \{x, y, z\})\\
&F_{Var}(P, v) = F_{VarPre}(P, v) \to F_{VarPost}(P, v)\\
&initials(F_{VarPost}) \cap vs = \varnothing
\end{aligned}$$

provided

$$\begin{aligned}
&x, y, z \in v\\
&vs = \{\mid get_1, \cdots, get_m, set_1, \cdots, set_m, end \mid\}
\end{aligned}$$

B.1.2.2 F_{mrg}

Definition B.1.4 (F_{mrg}). *A $F_{mrg}(F_{VarPre}(P_1), F_{VarPre}(P_2))$ function is defined to merge the prefixes of $F_{Var}(P_1)$ and $F_{Var}(P_2)$ into one final prefix.*

- *Basically, it is equal to $F_{VarPre}(P_1) \to F_{VarPre}(P_2)$ if both $F_{Var}(P_1)$ and $F_{Var}(P_2)$ are not empty and do not share any local variables in its get event list. However, if $F_{Var}(P_2)$ returns the same variables which have been returned by $F_{Var}(P_1)$, then the get events to return these same variables are removed from $F_{Var}(P_2)$.*
- *If exact one of $F_{Var}(P_1)$ and $F_{Var}(P_2)$ is empty, then it is equal to the non-empty one.*
- *If both $F_{Var}(P_1)$ and $F_{Var}(P_2)$ are empty, then it is equal to empty.*

For example,

$$\begin{aligned}
&F_{mrg}(get_i?l_i, get_j?l_j) = get_i?l_i \to get_j?l_j\\
&F_{mrg}(get_i?l_i, get_j?l_j \to get_i?l_i) = get_i?l_i \to get_j?l_j\\
&F_{mrg}(, get_j?l_j \to get_i?l_i) = get_j?l_j \to get_i?l_i\\
&F_{mrg}(get_i?l_i,) = get_i?l_i\\
&F_{mrg}(,) =
\end{aligned}$$

B.1.2.3 F_{Var} Rules

F_{Var} Rule 1 (Unit). If all variables in v do not occur in P, then it is the same as P.

$$F_{Var}(P, v) = P$$

provided

$$scpV(P) \cap v = \varnothing$$

□

F_{Var} Rule 2 (Basic).

$$\begin{aligned}
&F_{Var}(SKIP, v) = SKIP\\
&F_{Var}(STOP, v) = STOP\\
&F_{Var}(\mathbf{div}, v) = \mathbf{div}
\end{aligned}$$

□

F_{Var} **Rule 3 (Prefix).**

$$\begin{aligned}
&F_{Var}\left(c \rightarrow P, \{\}\right) = c \rightarrow F_{Var}\left(P\right)\\
&F_{Var}\left(c?x \rightarrow P, \{x\}\right) = c?x \rightarrow set_i!x \rightarrow F_{Var}\left(P\right)\\
&F_{Var}\left(c?x \rightarrow P, \{\}\right) = c?x \rightarrow F_{Var}\left(P\right)\\
&F_{Var}\left(c!x \rightarrow P, \{x\}\right) = get_i?x \rightarrow c!x \rightarrow F_{Var}\left(P\right)\\
&F_{Var}\left(c.x \rightarrow P, \{x\}\right) = get_i?x \rightarrow c.x \rightarrow F_{Var}\left(P\right)\\
&F_{Var}\left(c?x?y : a!z \rightarrow P, \{x, z\}\right) = get_i?z \rightarrow c?x?y : a!z \rightarrow set_j!x \rightarrow F_{Var}\left(P\right)
\end{aligned}$$

□

F_{Var} **Rule 4 (Sequential Composition).**

$$F_{Var}\left(P \,;\, Q, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(F_{VarPost}\left(P, v\right) ; F_{Var}\left(Q, v\right)\right)$$

□

F_{Var} **Rule 5 (Hiding).**

$$F_{Var}\left(P \setminus a, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(F_{VarPost}\left(P, v\right) \setminus a\right)$$

□

F_{Var} **Rule 6 (External Choice).**

$$\begin{aligned}
&F_{Var}\left(P \,\Box\, Q, v\right)\\
&= F_{mrg}\left(F_{VarPre}\left(P, v\right), F_{VarPre}\left(Q, v\right)\right) \rightarrow \left(F_{VarPost}\left(P, v\right) \Box F_{VarPost}\left(Q, v\right)\right)
\end{aligned}$$

If the first construct of P and Q does not evaluate any variables in v, or both F_{VarPre} (P, v) and $F_{VarPre}\left(Q, v\right)$ are empty,

$$\begin{aligned}
&initRdLclV\left(P\right) \cap v = \varnothing\\
&initRdLclV\left(Q\right) \cap v = \varnothing
\end{aligned}$$

then it is simplified to

$$F_{Var}\left(P \,\Box\, Q, v\right) = F_{Var}\left(P, v\right) \Box F_{Var}\left(Q, v\right)$$

□

F_{Var} **Rule 7 (Internal Choice).**

$$\begin{aligned}
&F_{Var}\left(P \sqcap Q, v\right)\\
&= F_{mrg}\left(F_{VarPre}\left(P, v\right), F_{VarPre}\left(Q, v\right)\right) \rightarrow \left(F_{VarPost}\left(P, v\right) \sqcap F_{VarPost}\left(Q, v\right)\right)
\end{aligned}$$

□

F_{Var} **Rule 8 (Boolean Guard).**

$$\begin{aligned}
&F_{Var}\left(b \;\&\, P, v\right) = F_{mrg}\left(F_{VarPre}\left(b, v\right), F_{VarPre}\left(P, v\right)\right) \rightarrow \left(b \;\&\, F_{VarPost}\left(P, v\right)\right)\\
&F_{VarPre}\left(b \;\&\, P, v\right) = F_{mrg}\left(F_{VarPre}\left(b, v\right), F_{VarPre}\left(P, v\right)\right)\\
&F_{VarPost}\left(b \;\&\, P, v\right) = \left(b \;\&\, F_{VarPost}\left(P, v\right)\right)
\end{aligned}$$

□

For example,

$$F_{Var}\left(l_p > 0\ \&\ c!l_q \rightarrow SKIP, \{l_p, l_q\}\right)$$
$$= get_p?l_p \rightarrow get_q?l_q \rightarrow \left(l_p > 0\ \&\ c!l_q \rightarrow SKIP\right)$$

F_{Var} **Rule 9 (Parallel Composition).**

$$F_{Var}\left(P \underset{cs}{\parallel} Q, v\right)$$
$$= F_{mrg}\left(F_{VarPre}\left(P, v\right), F_{VarPre}\left(Q, v\right)\right) \rightarrow \left(F_{VarPost}\left(P, v\right) \underset{cs}{\parallel} F_{VarPost}\left(Q, v\right)\right)$$

□

F_{Var} **Rule 10 (Interleave).**

$$F_{Var}\left(P \mid\mid\mid Q, v\right)$$
$$= F_{mrg}\left(F_{VarPre}\left(P, v\right), F_{VarPre}\left(Q, v\right)\right) \rightarrow \left(F_{VarPost}\left(P, v\right) \mid\mid\mid F_{VarPost}\left(Q, v\right)\right)$$

□

F_{Var} **Rule 11 (Recursion).**

$$F_{Var}\left(\text{let } X = P\left(X\right) \text{ within } X, v\right) = \text{let } X = F_{Var}\left(P\left(X\right), v\right) \text{ within } X$$

□

F_{Var} **Rule 12 (Replicated Sequential Composition).**

$$F_{Var}\left(;x : a \bullet P, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(;x : a \bullet F_{VarPost}\left(P, v\right)\right)$$

□

F_{Var} **Rule 13 (Replicated External Choice).**

$$F_{Var}\left(\Box\, x : a \bullet P, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(\Box\, x : a \bullet F_{VarPost}\left(P, v\right)\right)$$

□

F_{Var} **Rule 14 (Replicated Internal Choice).**

$$F_{Var}\left(\sqcap\, x : a \bullet P, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(\sqcap\, x : a \bullet F_{VarPost}\left(P, v\right)\right)$$

□

F_{Var} **Rule 15 (Replicated Interleave).**

$$F_{Var}\left(\mid\mid\mid\, x : a \bullet P, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(\mid\mid\mid\, x : a \bullet F_{VarPost}\left(P, v\right)\right)$$

□

F_{Var} **Rule 16 (Replicated Parallel Composition).**

$$F_{Var}\left(\underset{cs}{\parallel}\ x : a \bullet P, v\right) = F_{VarPre}\left(P, v\right) \rightarrow \left(\underset{cs}{\parallel}\ x : a \bullet F_{VarPost}\left(P, v\right)\right)$$

□

B.1.3 Memory Mode of Process

Definition B.1.5 (**F_{Mem}**). *The function F_{Mem} gives a memory model for a CSP process P to store and retrieve local variables, which are shown in a set v with m elements: $l_1, \cdots, l_m$. F_{Mem} is only used in the translation of variable block in Φ Rule 38 and local variables v are introduced by replicated internal choice $\bigsqcap_{v: T_v}$.*

$$\begin{aligned} F_{Mem}(P, v) &= \left((Us \,;\, F_{Var}(P, v) \,;\, end \to SKIP) \underset{vs}{\parallel} Rep_{Mem} \right) \setminus vs \\ &= ((Us \,;\, F_{Var}(P, v) \,;\, end \to SKIP) \mathbin{/\!\!/}_{vs} Rep_{Mem}) \end{aligned}$$

where

$$Us = set_1!l_1 \to \cdots \to set_m!l_m \to SKIP$$

$$vs = \{\!|set_1, get_1, set_2, get_2, \cdots, set_m, get_m, end|\!\}$$

$$Rep_{Mem} = \underset{\{\!|end|\!\}}{\parallel} \{MemCell_i \mid i \in \{1..m\}\}$$

Rep_{Mem} is a replicated parallel composition of a set of MemCell processes

$$\{MemCell_1, \cdots, MemCell_m\}$$

that synchronise on the end event. It is equal to

$$\underset{\{\!|end|\!\}}{\parallel} i : 1 \,..\, m \bullet MemCell_i$$

According to Definition B.1.1 and Definition B.1.2, we can easily get

$$\begin{aligned} &\alpha(Us \,;\, F_{Var}(P, v) \,;\, end \to SKIP) = \alpha(P) \cup vs \\ &\alpha(Rep_{Mem}) = vs \end{aligned}$$

therefore

$$\alpha(Rep_{Mem}) \subseteq \alpha(Us \,;\, F_{Var}(P, v) \,;\, end \to SKIP)$$

Finally, the memory model can be rewritten to a process defined on the subordination [7] or enslavement [71] operator $/\!\!/$.

Actually, in $MemCell_i$ the i is a part of its name and not the parameter. So it should be like $strcat(MemCell, \_, [1])$, $strcat(MemCell, \_, [2])$, $\ldots$, and $strcat(MemCell, \_, [m])$. However, in order to simplify the typing in the formula, we still use $MemCell_i$ to represent it has a parameter i.

The F_{Mem} function has two input parameters: a CSP process P and a set of local variables v. Then it returns a process in which each variable (m variables) in P to be updated and retrieved by putting them in parallel with a set of $MemCell_i$ processes (Rep_{Mem}), and finally these additional communications are hidden from its environment.

Us is a process to update all m variables in v to their arbitrary values. vs is a set of all events that are used to set and get m variables in P as well as the end event. Rep_{Mem} is a replicated parallel composition of all $MemCell_i$ processes with the end event, which provides a way for all these processes along with P to terminate successfully.

B.2 Functions

B.2.1 α Function

Definition B.2.1 (α function). *The α function gives the alphabet of a theory or an object being studied.*

For an alphabetised predicate P in UTP, αP denotes a set of variable names to be studied, such as $okay$, $wait$, tr, ref and their dashed counterparts for reactive processes.

For a CSP process, αP denotes the alphabet of the process P, a set of names of events that are relevant for P. For example, $\alpha\left(c \rightarrow SKIP \Box d \rightarrow STOP\right) = \{c, d\}$.

And for a schema in Circus or Z, $\alpha\left(Schema\right)$ gives a set of components of the schema. For example,

$$\begin{aligned} \alpha\left(St\right) &= \{s_1, s_2\} \\ \alpha\left(St'\right) &= \{s_1', s_2'\} \end{aligned}$$

provided

$$St \mathrel{\widehat=} [\, s_1, s_2 : \mathbb{N} \mid true \,]$$

B.2.2 FV

Definition B.2.2 (FV function). *The FV function returns a set of free variables in a predicate or expression. For example,*

$$FV\left(\forall x : \mathbb{N} \bullet y > x\right) = \{y\}$$

B.2.3 DFV

Definition B.2.3 (DFV function). *The DFV function returns a set of dashed free variables in a predicate or expression. For example,*

$$\begin{aligned} &DFV\left(x' > 0 \land y = 0\right) = \{x'\} \\ &DFV\left(\forall x : \mathbb{N} \bullet y > x\right) = \{\} \end{aligned}$$

B.2.4 $UDFV$

Definition B.2.4 ($UDFV$ function). *The $UDFV$ function returns a set of undashed free variables in a predicate or expression. For example,*

$$\begin{aligned} &UDFV\left(x' > 0 \land y = 0\right) = \{y\} \\ &UDFV\left(\forall x : \mathbb{N} \bullet y > x\right) = \{y\} \end{aligned}$$

B.2.5 $wrtV$

Definition B.2.5 ($wrtV$ function). *The $wrtV$ function gives a set of variables written by an action or a CSP process. And if it is a CSP process, it denotes a set of local variables that have corresponding memory processes as defined in Definition B.1.1. For example,*

$$\begin{aligned} wrtV\left(SExp\right) &= \{s_2\} \\ wrtV\left(x, y := 1, z\right) &= \{x, y\} \end{aligned}$$

provided

$$\begin{aligned} &St \mathrel{\widehat=} [\, s_1, s_2 : \mathbb{N} \mid true \,] \\ &SExp \mathrel{\widehat=} [\, \Delta St \mid s_2' = s_1 \,] \end{aligned}$$

B.2.6 *usedV*

Definition B.2.6 (*usedV* function). *The usedV function returns a set of read-only variables, either state and local variables, in an action or a CSP process. And if it is a CSP process, it denotes a set of local variables that have corresponding memory processes as defined in Definition B.1.1. For example,*

$$\begin{aligned} usedV\,(SExp) &= \{s_1\} \\ usedV\,(x, y := 1, z) &= \{z\} \end{aligned}$$

provided

$$St \mathrel{\widehat=} [\, s_1, s_2 : \mathbb{N} \mid true \,]$$
$$SExp \mathrel{\widehat=} [\, \Delta St \mid s_2' = s_1 \,]$$

B.2.7 *usedC*

Definition B.2.7 (*usedC* function). *The usedC function returns a set of channels in an action or a process. For example,*

$$\begin{aligned} usedC\,(\mathbf{Skip}) &= \varnothing \\ usedC\,(c \rightarrow \mathbf{Skip}) &= \{c\} \\ usedC\,(c \rightarrow d \rightarrow \mathbf{Skip}) &= \{c, d\} \end{aligned}$$

B.2.8 *initials*

Definition B.2.8 (*initials* function). *The initials function gives a set of channels which an action may communicate first.*

$$\begin{aligned} initials\,(c \rightarrow \mathbf{Skip}) &= \{c\} \\ initials\,(c \rightarrow \mathbf{Skip} \mathbin{\Box} d \rightarrow \mathbf{Skip}) &= \{c, d\} \end{aligned}$$

B.2.9 *scpV*

Definition B.2.9 (*scpV* function). *The scpV function gives a set of free variables in scope in an action or a CSP process. It is equal to the union of wrtV and usedV.*

$$scpV\,(c?x!y \rightarrow \mathbf{Skip}) = \{x, y\}$$

B.2.10 *stV*

Definition B.2.10 (*stV* function). *The stV function gives a set of state components in an action. For example,*

$$\begin{aligned} stV\,(SExp) &= \{s_1, s_2\} \\ stV\,(s_1, l_1 := 1, s_2) &= \{s_1, s_2\} \end{aligned}$$

provided

$$SExp = [\Delta StPar \,;\, in? : T_i \,;\, out! : T_o \mid s_1' = in? \wedge out! = s_2']$$

B.2.11 *lclV*

Definition B.2.11 (*lclV* function). *The lclV function gives a set of local variables in an action or a CSP process or an expression. In a CSP process, it denotes the variables that have corresponding memory processes as defined in Definition B.1.1.*

$$lclV\,(SExp) = \{in, out\}$$
$$lclV\,(s_1, l_1 := l_2, s_2) = \{l_1, l_2\}$$

provided

$$SExp = [\Delta StPar\ ;\ in? : T_i\ ;\ out! : T_o \mid s_1' = in? \land out! = s_2']$$

B.2.12 *initStV*, *initRdStV*, and *initWrtStV*

Definition B.2.12 (*initStV*, *initRdStV*, and *initWrtStV* functions). *The initStV function gives a set of state components in the first construct of an action. And the initRdStV and initWrtStV are a set of state components to be read and written respectively in the first construct of an action. Particularly,*

$$initStV\,(A) = initRdStV\,(A) \cup initWrtStV\,(A)$$

For example,

$$initStV\,(SExp) = \varnothing$$
$$initRdStV\,(s_1, l_1 := 1, s_2) = \{s_2\}$$
$$initWrtStV\,(s_1, l_1 := 1, s_2) = \{s_1\}$$
$$initStV\,(s_1, l_1 := 1, s_2) = \{s_1, s_2\}$$
$$initRdStV\,(c.(s_1 + s_2)) = \{s_1, s_2\}$$
$$initWrtStV\,(c.(s_1 + s_2)) = \varnothing$$
$$initStV\,(c.(s_1 + s_2)) = \{s_1, s_2\}$$

B.2.13 *initLclV*, *initRdLclV*, and *initWrtLclV*

Definition B.2.13 (*initLclV*, *initRdLclV*, and *initWrtLclV* functions). *The initLclV function gives a set of local variables in the first construct of an action or a CSP process. For a CSP process, it denotes the variables that have corresponding memory processes as defined in Definition B.1.1. And the initRdLclV and initWrtLclV are a set of local variables to be read and written respectively in the first construct of an action. Particularly,*

$$initLclV\,(A) = initRdLclV\,(A) \cup initWrtLclV\,(A)$$

For example,

$$initRdLclV\,(SExp) = \{in\}$$
$$initWrtLclV\,(SExp) = \{out\}$$
$$initLclV\,(SExp) = \{in, out\}$$
$$initRdLclV\,(s_1, l_1 := l_2, s_2) = \{l_2\}$$
$$initWrtLclV\,(s_1, l_1 := l_2, s_2) = \{l_1\}$$
$$initLclV\,(s_1, l_1 := l_2, s_2) = \{l_1, l_2\}$$
$$initRdLclV\,(c.(l_1 + s_2)?l_2) = \{l_1\}$$
$$initWrtLclV\,(c.(l_1 + s_2)?l_2) = \{l_2\}$$
$$initLclV\,(c.(l_1 + s_2)?l_2) = \{l_1, l_2\}$$

provided

$$SExp = [\Delta StPar\ ;\ in? : T_i\ ;\ out! : T_o \mid s_1' = in? \land out! = s_2']$$

B.2.14 Renaming Function F_{Ren}

Definition B.2.14 (F_{Ren}). *The $F_{Ren}(term, \{(v_{old}, v_{new})\})$ function renames every free occurrence of variables in v_{old} in the term (such as action, expression or predicate) to the corresponding names in v_{new}. It is equal to the definition of the rule of substitution in First-Order Logic: $t[e/x]$ denotes the result of replacing all* free *occurrences of the variable x in the formula t by the term e; $t[e, f/x, y]$ is a multiple substitution of terms e and f for variables x and y in t. For example,*

$$F_{Ren}(x := y, \{(x, x_1), (y, y_1)\}) = x_1 := y_1$$
$$F_{Ren}(x := y, \{(y, y_1)\}) = x := y_1$$
$$F_{Ren}(\exists x : T_x \bullet (x := y), \{(y, y_1), (x, x_1)\}) = x := y_1$$
$$F_{Ren}(x := y, \{(y, y_1), (x, x_1)\}) = x_1 := y_1$$

B.2.15 Body Function - B

Definition B.2.15 (Body Function B). *The function B gives the body of an action or a process.*

B.2.16 Text/String Concatenation - $strcat$

Definition B.2.16 (Text/String Concatenation - $strcat$). *The $strcat(a, b, c)$ function is for string or text concatenation of a, b and c to form a new text abc. And $strcat(a, b, [c_1, c_2, \cdots, c_n]$ is for string concatenation to form $abc_1bc_2b \cdots bc_n$. For example,*

$$strcat(c, , 1) = c1$$
$$strcat(c, \_, 1) = c\_1$$
$$strcat(c, -, 1) = c - 1$$
$$strcat(c, \_, [x, y, z]) = c\_x\_y\_z$$

B.3 Prefixed Actions

Definition B.3.1 (Prefixed Actions). *A Circus action A is a* prefixed *action AA if its first construct is an event regardless of whether the event is external or internal. The reason to define* prefixed *actions is because external choice of actions can be resolved only by events or termination, and state changes will not resolve it according to its semantics. Fundamentally, basic actions are not prefixed actions. However, to simplify the actions occurred in external choice of actions, the prefixed actions are extended to include basic actions and finally basic actions are regarded as AA as well. Therefore, a* prefixed *action is safe to be an action in external choice when the external choice is linked to CSP by Φ. This definition is very similar to [54, Definition C.1] except basic actions.*

AA can be one of actions below.

- *Basic actions:* **Skip**, **Stop**, *or* **Chaos**.
- *Prefixing: $c \rightarrow A$ where c denotes all communication patterns allowed in Circus.*
- *Guarded action: $(g) \& AA$.*
- *Sequential composition: $AA ; A$.*
- *External choice: $AA_1 \Box AA_2$.*
- *Internal choice: $AA_1 \sqcap AA_2$.*

- *Parallel composition:* $AA_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket AA_2$.
- *Interleaving:* $AA_1 \;|||[\; ns_1 \mid ns_2 \;]|||\; AA_2$.
- *Hiding:* $AA \setminus cs \quad \text{provided} \quad initials(AA) \cap cs = \varnothing$.
- *Recursion:* $\mu X \bullet AA(X)$. *Particularly, the first channel event of* $AA(X)$ *does not evaluate state variables. That is,* $initRdStV\,(AA) = \varnothing$.
- *Action invocation A:* $A \mathrel{\widehat=} AA$.
- *Parametrised action:* $A \mathrel{\widehat=} x : T \bullet AA(x)$.
- *Iterated sequential composition:* $;\, x : T \bullet AA(x)$.
- *Iterated external choice:* $\Box\, x : T \bullet AA(x)$.
- *Iterated internal choice:* $\bigsqcap x : T \bullet AA(x)$.
- *Iterated parallel composition:*$\llbracket cs \rrbracket\, x : T \llbracket ns \rrbracket \bullet AA(x)$.
- *Iterated interleaving:* $|||\, x : T \;|||[\; ns \;]|||\; \bullet AA(x)$.
- *Alternation:* $\mathbf{if}\ g_1 \longrightarrow AA_1 \;[\!]\; \cdots \;[\!]\; g_n \longrightarrow AA_n\ \mathbf{fi}$ *provided all actions are AA.*
- *Alternation (mutual exclusive guards):* $\mathbf{if}\ g_1 \longrightarrow A_1 \;[\!]\; \cdots \;[\!]\; g_n \longrightarrow A_n\ \mathbf{fi}$ *provided exactly one guard is evaluated to be true and others to be false at the same time.*
- *Variable block:* $\mathbf{var}\ x : T \bullet AA(x)$.
- *Renaming:* $AA[v_{old} := v_{new}]$.

B.4 Prefixed Processes

Definition B.4.1 (Prefixed Processes). *A Circus process P is a* prefixed *process PA if its main action is a AA and the first event of AA does not evaluate state variables of P. That is,* $initRdStV\,(PA.A) = \varnothing$. *Therefore,* $R_{pre}\,(PA.A)$ *is empty. Finally,*

$$R_{wrt}\,(PA.A) = R_{post}\,(PA.A) \qquad \text{[Definition 4.3.1]}$$

PA can be one of process below.

- *Explicitly defined process with its main action AA*

```
process  PA ≙ begin
    state StPar == ···
    Pars == [···]
    • AA
end
```

- *Sequential composition:* $PA\,;P$
- *Internal choice:* $PA_1 \sqcap PA_2$.
- *External choice:* $PA_1 \Box PA_2$.
- *Parallel composition:* $PA_1 \llbracket cs \rrbracket PA_2$
- *Interleaving:* $PA_1 \mathbin{|||} PA_2$

- *Hiding:* $PA \setminus cs$
- *Iterated sequential composition:* $\mathop{;}\ x : T \bullet PA(x)$
- *Iterated external choice:* $\Box\ x : T \bullet PA(x)$.
- *Iterated internal choice:* $\bigsqcap\ x : T \bullet PA(x)$.
- *Iterated parallel composition:*$[\![CS]\!]\ x : T \bullet PA(x)$
- *Iterated interleaving:* $|||\ x : T \bullet PA(x)$.
- *Parametrised process:* $x : T \bullet PA$.
- *Indexed process:* $x : T \odot PA$.
- *Process invocation:* PA *or* $PA\,(e)$
- *Process renaming:* $PA[c_{old} := c_{new}]$

Appendix C

Theorems, Lemmas and Laws

This appendix lists lemmas, laws, and theorems that are used in Chapter 5.

C.1 F_{Mem}

This section aims to prove Theorem C.1.28 which is based on a collection of lemmas in Section C.1.2. In order to prove these lemmas, firstly a number of laws are given and proved.

In this section, we use a number of abbreviations given below.

Definition C.1.1 (Abbreviations).

$$Rep_{Mem} = \left(\underset{\{\!|end|\!\}}{\|} i : 1..m \bullet MemCell_i \right)$$

$$Rep_{MCell} = \left(\underset{\{\!|end|\!\}}{\|} i : 1..m \bullet (MCell_i\,(x_i)) \right)$$

$$P \mathbin{/\!\!/}_{vs} Q = \left(P \underset{vs}{\|} Q \right) \setminus vs$$

C.1.1 Laws

C.1.1.1 CP and Proof Strategies

A number of laws about the memory model of a CSP process are presented and proved in this section. For some laws, they are applied to general processes. And but for others, in order to prove them, we only take into account CSP processes that are the target processes of our link rules. According to Appendix E, all CSP processes considered are listed below and denoted as CP.

- Basic processes: $SKIP$, $STOP$, and **div**
- Prefixing: $c \to CP$
- Sequential composition: $CP_1 \mathbin{;} CP_2$
- Boolean guard: $b \;\&\; CP$
- External choice: $CP_1 \,\Box\, CP_2$, and both actions are not $SKIP$
- Internal choice: $CP_1 \sqcap CP_2$
- Hiding: $CP \setminus cs$

- Recursion: let $X = CP(X)$ within X
- Parallel composition: $CP_1 \underset{cs}{\parallel} CP_2$
- Interleaving: $CP_1 \mathbin{|||} CP_2$
- Replicated sequential composition: $;x : a \bullet CP$
- Replicated external choice: $\Box\, x : a \bullet CP$
- Replicated internal choice: $\bigsqcap x : a \bullet CP$
- Replicated parallel composition: $\underset{cs}{\Big\|}\ x : a \bullet CP$
- Replicated interleaving: $\Big|\Big|\Big|\, x : a \bullet CP$

And afterwards, we use P and Q to denote general CSP processes and CP to denote the processes above.

Furthermore, it is difficult to prove a law for all combinations of CP. For example, if a law about the memory model of external choice is to be proved, it is very hard to prove this law valid for all combinations of CP_1 and CP_2 since there are a large number of combinations. In order to simplify the proof, our solution is to abstract all CP processes through a *Abs* function according to their initial events. This abstraction is applied mainly due to the step law of each process. For example, a parallel composition can be turned into a sequential process since they do not have fundamental distinction in CSP [71, p. 46]. And the *Abs* function defined for each CP is shown below. It is worth noting that recursion is not supported in the memory model. And replicated operations are just the expansion of the corresponding operators. Therefore they are omitted.

$$Abs\,(SKIP) = SKIP$$
$$Abs\,(STOP) = STOP$$
$$Abs\,(\mathbf{div}) = \mathbf{div}$$
$$Abs\,(c \to CP) = c \to CP$$
$$Abs\,(CP_1\,;\,CP_2) = \begin{cases} Abs\,(CP_2) & CP_1 = SKIP \\ Abs\,(CP_1) & CP_1 \neq SKIP \end{cases}$$
$$Abs\,(b\ \&\ \ CP) = \begin{cases} Abs\,(CP) & b = true \\ STOP & b = false \end{cases}$$
$$Abs\,(CP_1 \Box CP_2) = \begin{cases} \mathbf{div} & CP_1 = \mathbf{div} \vee CP_2 = \mathbf{div} \\ Abs\,(CP_1) & CP_2 = STOP \\ Abs\,(CP_2) & CP_1 = STOP \\ Abs\begin{pmatrix} (CP_{11} \Box CP_2) \\ \sqcap \\ (CP_{12} \Box CP_2) \end{pmatrix} & CP_1 = CP_{11} \sqcap CP_{12} \\ Abs\begin{pmatrix} (CP_{21} \Box CP_1) \\ \sqcap \\ (CP_{22} \Box CP_1) \end{pmatrix} & CP_2 = CP_{21} \sqcap CP_{22} \\ ?x : A \to CP & \begin{pmatrix} initials\,(CP_1) \neq \varnothing \\ \vee \\ initials\,(CP_2) \neq \varnothing \end{pmatrix} \end{cases}$$

$$Abs\,(CP_1 \sqcap CP_2) = \begin{cases} \mathbf{div} & CP_1 = \mathbf{div} \vee CP_2 = \mathbf{div} \\ Abs\,(CP_1) \sqcap Abs\,(CP_2) & CP_1 \neq \mathbf{div} \wedge CP_2 \neq \mathbf{div} \end{cases}$$

$$Abs\,(CP \setminus cs) = \begin{cases} Abs\,(CP) & initials\,(CP) \cap cs = \varnothing \\ Abs\begin{pmatrix} (CP_1 \setminus cs) \\ \sqcap \\ (CP_2 \setminus cs) \end{pmatrix} & CP = CP_1 \sqcap CP_2 \\ Abs\,(CP_1) & (CP = ?x : A \to CP_1) \wedge (A \subseteq cs) \end{cases}$$

$$Abs\left(CP_1 \underset{cs}{\parallel} CP_2\right) = \begin{cases} \mathbf{div} & CP_1 = \mathbf{div} \vee CP_2 = \mathbf{div} \\ SKIP & CP_1 = SKIP \wedge CP_2 = SKIP \\ Abs\begin{pmatrix} \left(CP_{11} \underset{cs}{\parallel} CP_2\right) \\ \sqcap \\ \left(CP_{12} \underset{cs}{\parallel} CP_2\right) \end{pmatrix} & CP_1 = CP_{11} \sqcap CP_{12} \\ Abs\begin{pmatrix} \left(CP_{21} \underset{cs}{\parallel} CP_1\right) \\ \sqcap \\ \left(CP_{22} \underset{cs}{\parallel} CP_1\right) \end{pmatrix} & CP_2 = CP_{21} \sqcap CP_{22} \\ ?x : A \to CP & \begin{pmatrix} initials\,(CP_1) \neq \varnothing \\ \vee \\ initials\,(CP_2) \neq \varnothing \end{pmatrix} \end{cases}$$

$$Abs\,(CP_1 \mathbin{|||} CP_2) = \begin{cases} \mathbf{div} & CP_1 = \mathbf{div} \vee CP_2 = \mathbf{div} \\ SKIP & CP_1 = SKIP \wedge CP_2 = SKIP \\ Abs\begin{pmatrix} (CP_{11} \mathbin{|||} CP_2) \\ \sqcap \\ (CP_{12} \mathbin{|||} CP_2) \end{pmatrix} & CP_1 = CP_{11} \sqcap CP_{12} \\ Abs\begin{pmatrix} (CP_{21} \mathbin{|||} CP_1) \\ \sqcap \\ (CP_{22} \mathbin{|||} CP_1) \end{pmatrix} & CP_2 = CP_{21} \sqcap CP_{22} \\ ?x : A \to CP & \begin{pmatrix} initials\,(CP_1) \neq \varnothing \\ \vee \\ initials\,(CP_2) \neq \varnothing \end{pmatrix} \end{cases}$$

From abstraction, we can conclude that the processes after reduction have the following forms.

- Basic processes: $SKIP$, $STOP$, and **div**
- Prefixing: $?x : A \to CP$
- Internal choice: $CP_1 \sqcap CP_2$

Therefore, in order to prove a law is valid for all *CP* processes, we only consider the combinations of these reduced processes, instead of all *CP* processes. Eventually, the proof is simplified.

C.1.1.2 Deadlock

Law C.1.1 (Deadlock).

$$(STOP \mathbin{/\!/}_{vs} Rep_{MCell}) = STOP$$

Proof.

$$\begin{aligned}
&(STOP \mathbin{/\!/}_{vs} Rep_{MCell})\\
&= \left(\begin{array}{l} STOP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i))\right)\end{array}\right) \setminus vs && \text{[Abbreviations C.1.1]}\\
&= (STOP) \setminus vs && \text{[Hoare [7, Section 2.3.1, L3A]]}\\
&= STOP && \text{[Hoare [7, Section 3.5.1, L4]]}
\end{aligned}$$

□

C.1.1.3 Divergence

Law C.1.2 (Divergent Process).

$$(\mathbf{div} \mathbin{/\!/}_{vs} Rep_{MCell}) = \mathbf{div}$$

Proof.

$$\begin{aligned}
&(\mathbf{div} \mathbin{/\!/}_{vs} Rep_{MCell})\\
&= \left(\begin{array}{l} \mathbf{div} \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i))\right)\end{array}\right) \setminus vs && \text{[Abbreviations C.1.1]}\\
&= (\mathbf{div}) \setminus vs && \text{[Hoare [7, Section 3.8.1, L2]]}\\
&= \mathbf{div} && \text{[Hoare [7, Section 3.8.1, L2]]}
\end{aligned}$$

□

C.1.1.4 Get the value of a local variable

Law C.1.3 (Get the value of a local variable). *Provided the current values of local variables $l_1, \cdots, l_m$ are $x_1, \cdots, x_m$, then get_p will return the current value x_p of l_p.*

$$\begin{aligned}
&((get_p?l_p \to P\,(l_p)) \mathbin{/\!/}_{vs} Rep_{MCell})\\
&= (P\,(x_p) \mathbin{/\!/}_{vs} Rep_{MCell})
\end{aligned}$$

Therefore, after one-step of parallel, the value of l_p in the memory has been returned in P.

Proof.

$$((get_p?l_p \to P\,(l_p)) \mathbin{/\!/}_{vs} Rep_{MCell})$$

$$= \left(\begin{array}{l} (get_p?l_p \rightarrow P(l_p)) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \end{array} \right) \setminus vs \qquad \text{[Abbreviations C.1.1]}$$

$$= \left(\begin{array}{l} (P(x_p)) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \end{array} \right) \setminus vs \qquad \text{[Hoare [7, Section 4.5.1, L1B]]}$$

$$= (P\,(x_p) \mathbin{/\!\!/}_{vs} Rep_{MCell}) \qquad \text{[Abbreviations C.1.1]}$$

□

C.1.1.5 Set the value of a local variable

Law C.1.4 (Set the value of a local variable).

$$((set_p!l_p \rightarrow P) \mathbin{/\!\!/}_{vs} Rep_{MCell})$$
$$= \left(P\,(x_p) \mathbin{/\!\!/}_{vs} \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \begin{cases} MCell_i\,(x_i) & i \neq p \\ MCell_p\,(l_p) & i = p \end{cases} \right) \right)$$

For the memory cell of a local variable l_p, its value has been updated to l_p from the process.

Proof.

$$((set_p!l_p \rightarrow P) \mathbin{/\!\!/}_{vs} Rep_{MCell})$$

$$\left(\begin{array}{l} (set_p!l_p \rightarrow P) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \end{array} \right) \setminus vs \qquad \text{[Abbreviations C.1.1]}$$

$$= \left(\begin{array}{l} (P) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \begin{cases} MCell_i\,(x_i) & i \neq p \\ MCell_p\,(l_p) & i = p \end{cases} \right) \end{array} \right) \setminus vs$$

[Hoare [7, Section 4.5.1, L1A]]

$$= \left(P\,(x_p) \mathbin{/\!\!/}_{vs} \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \begin{cases} MCell_i\,(x_i) & i \neq p \\ MCell_p\,(l_p) & i = p \end{cases} \right) \right) \qquad \text{[Abbreviations C.1.1]}$$

□

C.1.1.6 External choice - distribution (1)

Law C.1.5 (External choice - distribution (1)).

$$((c \rightarrow P \,\Box\, d \rightarrow Q) \mathbin{/\!\!/}_{vs} Rep_{MCell})$$
$$= (c \rightarrow (P \mathbin{/\!\!/}_{vs} Rep_{MCell})) \,\Box\, (d \rightarrow (Q \mathbin{/\!\!/}_{vs} Rep_{MCell}))$$

provided

- $c \notin vs$
- $d \notin vs$

Proof. According to [71, Section 1.2.1], the external choice of processes $(c \rightarrow P \Box d \rightarrow Q)$ is exactly the same as the guarded alternative $(c \rightarrow P \mid d \rightarrow Q)$ which can be rewritten as the prefix-choice $?x : \{c, d\} \rightarrow PP(x)$ where

$$PP(x) = \begin{cases} P & \text{if } x = c \\ Q & \text{if } x = d \end{cases}$$

Then

$$\begin{aligned}
&((c \rightarrow P \Box d \rightarrow Q) \mathbin{/\!\!/}_{vs} Rep_{MCell}) \\
&=?x : \{c, d\} \rightarrow (PP(x) \mathbin{/\!\!/}_{vs} Rep_{MCell}) \\
&\hfill [\text{Roscoe [71, } \langle \| \text{ -step} \rangle (3.10)] \text{ and } c \notin vs \wedge d \notin vs\ ] \\
&= (c \rightarrow (P \mathbin{/\!\!/}_{vs} Rep_{MCell})) \Box (d \rightarrow (Q \mathbin{/\!\!/}_{vs} Rep_{MCell})) \\
&\hfill [\text{Roscoe [71, Prefix-choice to External choice]}]
\end{aligned}$$

This law is also valid if c is the same event as d. □

C.1.1.7 External choice - distribution

Law C.1.6 (External choice - distribution).

$$\begin{aligned}
&(CP_1 \Box CP_2) \mathbin{/\!\!/}_{vs} Rep_{MCell} \\
&= (CP_1 \mathbin{/\!\!/}_{vs} Rep_{MCell}) \Box (CP_2 \mathbin{/\!\!/}_{vs} Rep_{MCell})
\end{aligned}$$

provided *both CP_1 and CP_2 are not $SKIP$, and*

$$\begin{aligned}
&initials\,(CP_1) \cap vs = \varnothing \\
&initials\,(CP_2) \cap vs = \varnothing
\end{aligned}$$

The side conditions are reasonable for the memory model because all events from vs have been moved out of external choice by F_{Var} Rule 6.

Proof. CP denotes CSP processes that are used in our link rules and are given in Section C.1.1.1. To prove this law is valid, we need to consider all combinations of abstracted CP processes that are described in Section C.1.1.1. However, according to [71, $\langle \Box$ -dist$\rangle$(2.7)], external choice $\Box$ is symmetric. Therefore, the combination of CP_1 as a form F_1 and CP_2 as another form F_2 has similar proof to the combination of CP_1 as a form F_2 and CP_2 as another form F_1. Finally, we only need to consider one of them.

(1) If CP_1 is $STOP$, then the LHS of the law

$$\begin{aligned}
&(CP_1 \Box CP_2) \mathbin{/\!\!/}_{vs} Rep_{MCell} \\
&= (STOP \Box CP_2) \mathbin{/\!\!/}_{vs} Rep_{MCell} \\
&= (CP_2) \mathbin{/\!\!/}_{vs} Rep_{MCell} && [\text{Hoare [7, Section 3.3.1, L4]}]
\end{aligned}$$

and the RHS of the law

$$\begin{aligned}
&(CP_1 \mathbin{/\!\!/}_{vs} Rep_{MCell}) \Box (CP_2 \mathbin{/\!\!/}_{vs} Rep_{MCell}) \\
&= (STOP \mathbin{/\!\!/}_{vs} Rep_{MCell}) \Box (CP_2 \mathbin{/\!\!/}_{vs} Rep_{MCell}) \\
&= STOP \Box (CP_2 \mathbin{/\!\!/}_{vs} Rep_{MCell}) && [\text{Law C.1.1}] \\
&= (CP_2) \mathbin{/\!\!/}_{vs} Rep_{MCell} && [\text{Hoare [7, Section 3.3.1, L4]}]
\end{aligned}$$

therefore, the law is valid.

(2) If CP_1 is **div**, then the LHS of the law

$$\begin{aligned}
&(CP_1 \Box CP_2) \parallel_{vs} Rep_{MCell} \\
&= (\mathbf{div} \Box CP_2) \parallel_{vs} Rep_{MCell} \\
&= (\mathbf{div}) \parallel_{vs} Rep_{MCell} && \text{[Hoare [7, Section 3.8.1, L2]]} \\
&= \mathbf{div} && \text{[Law C.1.2]}
\end{aligned}$$

and the RHS of the law

$$\begin{aligned}
&(CP_1 \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell}) \\
&= (\mathbf{div} \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell}) \\
&= \mathbf{div} \Box (CP_2 \parallel_{vs} Rep_{MCell}) && \text{[Law C.1.2]} \\
&= \mathbf{div} && \text{[Hoare [7, Section 3.8.1, L2]]}
\end{aligned}$$

therefore, the law is valid.

(3) If CP_1 is internal choice, assume

$$(CP_{11} \Box CP_2) \parallel_{vs} Rep_{MCell} = (CP_{11} \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell})$$

and

$$(CP_{12} \Box CP_2) \parallel_{vs} Rep_{MCell} = (CP_{12} \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell})$$

then the LHS of the law

$$\begin{aligned}
&((CP_{11} \sqcap CP_{12}) \Box CP_2) \parallel_{vs} Rep_{MCell} \\
&= ((CP_{11} \Box CP_2) \sqcap (CP_{12} \Box CP_2)) \parallel_{vs} Rep_{MCell} && \text{[Roscoe [71, } \langle \Box \text{ -dist} \rangle (2.7)]] \\
&= \begin{pmatrix} (CP_{11} \Box CP_2) \parallel_{vs} Rep_{MCell} \\ \sqcap \\ (CP_{12} \Box CP_2) \parallel_{vs} Rep_{MCell} \end{pmatrix} && \text{[Law C.1.9]}
\end{aligned}$$

the RHS of the law,

$$\begin{aligned}
&((CP_{11} \sqcap CP_{12}) \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell}) \\
&= ((CP_{11} \parallel_{vs} Rep_{MCell}) \sqcap (CP_2 \parallel_{vs} Rep_{MCell})) \Box (CP_2 \parallel_{vs} Rep_{MCell}) && \text{[Law C.1.9]} \\
&= \begin{pmatrix} (CP_{11} \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell}) \\ \sqcap \\ (CP_{12} \parallel_{vs} Rep_{MCell}) \Box (CP_2 \parallel_{vs} Rep_{MCell}) \end{pmatrix} && \text{[Roscoe [71, } \langle \Box \text{ -dist} \rangle (2.7)]] \\
&= \begin{pmatrix} (CP_{11} \Box CP_2) \parallel_{vs} Rep_{MCell} \\ \sqcap \\ (CP_{12} \Box CP_2) \parallel_{vs} Rep_{MCell} \end{pmatrix} && \text{[Assumption]}
\end{aligned}$$

therefore, the law is valid for the situation that one of processes is internal choice. In other words, that one of process in the external choice is an internal choice preserves the distribution of external choice in the memory model.

(4) If CP_1 is a prefixing and CP_2 is a prefixing as well, according to Law C.1.5 and conditions that initial events of CP_1 and CP_2 do not contain events from vs, this law is also valid.

Since this law is valid for all combinations of abstracted CP processes, this law is valid for all CP processes. □

C.1.1.8 External choice and Sequential composition - distribution (1)

Law C.1.7 (External choice and Sequential composition - distribution (1)).

$$\begin{array}{l}\left(\ ((c \to P \Box d \to Q) \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}\ \right) \\ = \left(\begin{array}{l}(c \to ((P \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell})) \\ \Box \\ (d \to ((Q \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}))\end{array}\right)\end{array}$$

provided

- $c \notin vs$
- $d \notin vs$

Proof.

$$\begin{array}{ll}\left(\ ((c \to P \Box d \to Q) \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}\ \right) & \\ = \left(\ ((?x : \{c, d\} \to PP(x)) \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}\ \right) & \\ \qquad\qquad\qquad [PP(x) \text{ definition in the proof of Law C.1.6}] & \\ = \left(\ (?x : \{c, d\} \to (PP(x) \mathbin{;} end \to SKIP)) \parallel_{vs} Rep_{MCell}\ \right) & \\ \qquad [\text{Roscoe [71, } \langle\text{;-step}\rangle\text{(7.4)] where } c \text{ and } d \text{ are not free in } (end \to SKIP)] & \\ = \left(\begin{array}{l}(c \to ((P \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell})) \\ \Box \\ (d \to ((Q \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}))\end{array}\right) & [\text{Law C.1.6}]\end{array}$$

□

C.1.1.9 External choice and Sequential composition - distribution

Law C.1.8 (External choice and Sequential composition - distribution).

$$\begin{array}{l}\left(\ ((CP_1 \Box CP_2) \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}\ \right) \\ = \left(\begin{array}{l}((CP_1 \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell})\end{array}\right)\end{array}$$

provided *both CP_1 and CP_2 are not $SKIP$, and*

$$\begin{array}{l} initials\,(CP_1) \cap vs = \varnothing \\ initials\,(CP_2) \cap vs = \varnothing \end{array}$$

The side conditions are reasonable for the memory model because all events from vs have been moved out of external choice by F_{Var} Rule 6.

Proof. The proof of this law is very similar to that of Law C.1.6.

(1) If CP_1 is $STOP$, then the LHS of the law

$$\begin{array}{ll}((CP_1 \Box CP_2) \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell} & \\ = ((STOP \Box CP_2) \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell} & \\ = (CP_2 \mathbin{;} end \to SKIP) \parallel_{vs} Rep_{MCell} & [\text{Hoare [7, Section 3.3.1, L4]}]\end{array}$$

and the RHS of the law

$$\begin{array}{ll}
\left(\begin{array}{l} ((CP_1 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \\
= \left(\begin{array}{l} ((STOP \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \\
= \left(\begin{array}{l} (STOP \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \text{[Hoare [7, Section 5.2, L5]]} \\
= \left(\begin{array}{l} STOP \,\Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \text{[Law C.1.1]} \\
= (CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} & \text{[Hoare [7, Section 3.3.1, L4]]}
\end{array}$$

therefore, the law is valid.

(2) If CP_1 is **div**, then the LHS of the law

$$\begin{array}{ll}
((CP_1 \,\Box\, CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} & \\
= ((\mathbf{div} \,\Box\, CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} & \\
= (\mathbf{div} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} & \text{[Hoare [7, Section 3.8.1, L2]]} \\
= (\mathbf{div}) \parallel_{vs} Rep_{MCell} & \text{[Hoare [7, Section 5.3.2, L1]]} \\
= \mathbf{div} & \text{[Law C.1.2]}
\end{array}$$

and the RHS of the law

$$\begin{array}{ll}
\left(\begin{array}{l} ((CP_1 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \\
= \left(\begin{array}{l} ((\mathbf{div} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \\
= \left(\begin{array}{l} (\mathbf{div} \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \text{[Hoare [7, Section 5.3.2, L1]]} \\
= \left(\begin{array}{l} \mathbf{div} \,\Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right) & \text{[Law C.1.2]} \\
= \mathbf{div} & \text{[Hoare [7, Section 3.8.1, L2]]}
\end{array}$$

therefore, the law is valid.

(3) If CP_1 is an internal choice, assume

$$\begin{array}{l}
\left(\; ((CP_{11} \,\Box\, CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \;\right) \\
= \left(\begin{array}{l} ((CP_{11} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right)
\end{array}$$

and

$$\begin{array}{l}
\left(\; ((CP_{12} \,\Box\, CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \;\right) \\
= \left(\begin{array}{l} ((CP_{12} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array}\right)
\end{array}$$

then the LHS of the law

$$
\begin{aligned}
&(((CP_{11} \sqcap CP_{12}) \Box CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \\
&= \left(\left(\begin{array}{l} (CP_{11} \Box CP_2) \\ \sqcap \\ (CP_{12} \Box CP_2) \end{array} \right) ; end \rightarrow SKIP \right) \parallel_{vs} Rep_{MCell} \\
&\hfill \text{[Roscoe [71, } \langle \Box\text{-dist} \rangle (2.7)]] \\
&= \left(\begin{array}{l} ((CP_{11} \Box CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \\ \sqcap \\ ((CP_{12} \Box CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \end{array} \right) \qquad \text{[Law C.1.10]}
\end{aligned}
$$

the RHS of the law,

$$
\begin{aligned}
&\left(\begin{array}{l} (((CP_{11} \sqcap CP_{12}) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array} \right) \\
&= \left(\begin{array}{l} \left(\begin{array}{l} ((CP_{11} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \sqcap \\ ((CP_{12} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array} \right) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array} \right) \qquad \text{[Law C.1.10]} \\
&= \left(\begin{array}{l} \left(\begin{array}{l} ((CP_{11} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array} \right) \\ \sqcap \\ \left(\begin{array}{l} ((CP_{12} \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \\ \Box \\ ((CP_2 \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell}) \end{array} \right) \end{array} \right) \qquad \text{[Roscoe [71, } \langle \Box\text{-dist} \rangle (2.7)]] \\
&= \left(\begin{array}{l} ((CP_{11} \Box CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \\ \sqcap \\ ((CP_{12} \Box CP_2) \,;\, end \rightarrow SKIP) \parallel_{vs} Rep_{MCell} \end{array} \right) \qquad \text{[Assumption]}
\end{aligned}
$$

therefore, the law is valid for the situation that one of processes is internal choice. In other words, that one of process in the external choice is an internal choice preserves the distribution of external choice and sequential composition in the memory model.

(4) If CP_1 is a prefixing and CP_2 is a prefixing as well, according to Law C.1.7 and conditions that initial events of CP_1 and CP_2 do not contain events from vs, this law is also valid.

Since this law is valid for all combinations of abstracted CP processes, this law is valid for all CP processes. □

C.1.1.10 Internal choice - distribution

Law C.1.9 (Internal choice - distribution).

$$
\begin{aligned}
&((P \sqcap Q) \parallel_{vs} Rep_{Mem}) \\
&= \left(\begin{array}{l} (P \parallel_{vs} Rep_{Mem}) \\ \sqcap \\ (Q \parallel_{vs} Rep_{Mem}) \end{array} \right)
\end{aligned}
$$

Proof.

$$((P \sqcap Q) \parallel_{vs} Rep_{Mem})$$

$$= \left(\begin{array}{l} (P \mathbin{/\!\!/_{vs}} Rep_{Mem}) \\ \sqcap \\ (Q \mathbin{/\!\!/_{vs}} Rep_{Mem}) \end{array} \right) \qquad \text{[Roscoe [71, } \langle \| \text{ -dist}\rangle\text{(3.12)]}$$

□

C.1.1.11 Internal choice and Sequential composition - distribution

Law C.1.10 (Internal choice and Sequential composition - distribution).

$$\begin{array}{l} (((P \sqcap Q) \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \\ = \left(\begin{array}{l} ((P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \\ \sqcap \\ ((Q \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \end{array} \right) \end{array}$$

Proof.

$$\begin{array}{l} (((P \sqcap Q) \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \\ = (((P \mathbin{;} end \to SKIP) \sqcap (Q \mathbin{;} end \to SKIP)) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \\ \hfill \text{[Hoare [7, Section 5.3.2, L2A]]} \\ = \left(\begin{array}{l} ((P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \\ \sqcap \\ ((Q \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{Mem}) \end{array} \right) \qquad \text{[Law C.1.9]} \end{array}$$

□

C.1.1.12 Boolean guard - step

Law C.1.11 (Boolean guard - step).

$$\begin{array}{l} (g \mathbin{\&} P) \mathbin{/\!\!/_{vs}} Rep_{MCell} \\ = g \mathbin{\&} (P \mathbin{/\!\!/_{vs}} Rep_{MCell}) \end{array}$$

Proof. The LHS of the law

$$\begin{array}{ll} (g \mathbin{\&} P) \mathbin{/\!\!/_{vs}} Rep_{MCell} & \\ = \begin{cases} P \mathbin{/\!\!/_{vs}} Rep_{MCell} & g = true \\ STOP \mathbin{/\!\!/_{vs}} Rep_{MCell} & g = false \end{cases} & \text{[Boolean guard is a shorthand [71, p. 14]]} \\ = \begin{cases} P \mathbin{/\!\!/_{vs}} Rep_{MCell} & g = true \\ STOP & g = false \end{cases} & \text{[Law C.1.1]} \end{array}$$

and the RHS of the law

$$\begin{array}{ll} g \mathbin{\&} (P \mathbin{/\!\!/_{vs}} Rep_{MCell}) & \\ = \begin{cases} P \mathbin{/\!\!/_{vs}} Rep_{MCell} & g = true \\ STOP & g = false \end{cases} & \text{[Boolean guard is a shorthand [71, p. 14]]} \end{array}$$

therefore the law is proved. □

C.1.1.13 Boolean guard and Sequential composition - step

Law C.1.12 (Boolean guard and Sequential composition - step).

$$(g \mathbin{\&} P \mathbin{;} end \to SKIP) = g \mathbin{\&} (P \mathbin{;} end \to SKIP)$$

Proof. The LHS of the law

$$
\begin{aligned}
&(g \mathbin{\&} P \mathbin{;} end \to SKIP) \\
&= \begin{cases} P \mathbin{;} end \to SKIP & g = true \\ STOP \mathbin{;} end \to SKIP & g = false \end{cases} && \text{[Boolean guard is a shorthand [71, p. 14]]} \\
&= \begin{cases} P \mathbin{;} end \to SKIP & g = true \\ STOP & g = false \end{cases} && \text{[Hoare [7, Section 5.2, L5]]}
\end{aligned}
$$

and the RHS of the law

$$
\begin{aligned}
&g \mathbin{\&} (P \mathbin{;} end \to SKIP) \\
&= \begin{cases} P \mathbin{;} end \to SKIP & g = true \\ STOP & g = false \end{cases} && \text{[Boolean guard is a shorthand [71, p. 14]]}
\end{aligned}
$$

therefore, the LHS is equal to the RHS. The law is proved. □

C.1.1.14 Boolean guard and Sequential composition - distribution

Law C.1.13 (Boolean guard and Sequential composition - distribution).

$$(g \mathbin{\&} P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell} \quad = g \mathbin{\&} ((P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell})$$

Proof.

$$
\begin{aligned}
&(g \mathbin{\&} P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell} \\
&= (g \mathbin{\&} (P \mathbin{;} end \to SKIP)) \mathbin{/\!\!/_{vs}} Rep_{MCell} && \text{[Law C.1.12]} \\
&= g \mathbin{\&} ((P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell}) && \text{[Law C.1.11]}
\end{aligned}
$$

□

C.1.1.15 Sequential composition - distribution

Law C.1.14 (Sequential composition - distribution). *The memory model of a sequential composition $P \mathbin{;} Q$ is equal to a sequential composition of the memory model of P and finally terminated, and the memory model of Q with another copy of memory but all updates from P are kept.*

$$
\begin{aligned}
&\left(\ (P \mathbin{;} Q) \mathbin{/\!\!/_{vs}} Rep_{MCell}\ \right) \\
&= \left(\begin{array}{l} \left(\ (P \mathbin{;} end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell}\ \right) \\ ; \\ \left(Q \mathbin{/\!\!/_{vs}} \left(\mathop{\big\|}_{\{\!|end|\!\}} i : 1..m \bullet \left(\begin{cases} MCell_i\,(x_i) & i \notin j..k \\ MCell_i\,(l_i) & i \in j..k \end{cases} \right) \right) \right) \end{array} \right)
\end{aligned}
$$

provided *P is a translated process from another process P' by F_{Var} and*

$$wrtV\,(P) \land l = \{l_j, \cdots, l_k\}$$

which denotes that all local variables to be updated in P are l_j, …, and l_k.

Proof. This law is proved from two aspects: the local variables aspect and the behavioural aspect.

(1) The LHS of the law has $P \mathbin{;} Q$ in a same memory, thus the memory that Q sees is the one after the update from P. And in the RHS of the law, P still starts with the

same memory as that in the LHS, but for Q its memory is another copy of the original memory but all updates from P are seen. Therefore, Q in the RHS also starts with the same memory as that in the LHS. Finally, from the local variables aspect, both LHS and RHS are the same.

(2) From the LHS of the law, if P terminates, then Q will begin to execute. From the RHS of the law, if P terminates, according to Lemma C.1.2 its memory model

$$(P \,;\, end \rightarrow SKIP) \,/\!\!/_{vs}\, Rep_{MCell}$$

terminates as well and finally the memory model of Q begins to execute. And if P does not terminate, both LHS and RHS will not terminate. Finally, from the behavioural aspect, both LHS and RHS are the same.

In sum, the law is valid. □

C.1.1.16 Hiding - combination

Law C.1.15 (Hiding - combination).

$$\begin{aligned}
&\left((P \setminus cs) \,/\!\!/_{vs}\, Rep_{MCell} \right)\\
&= \left(P \underset{vs}{\parallel} \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \right) \setminus (vs \cup cs)\\
&= \left(P \,/\!\!/_{vs}\, Rep_{MCell} \right) \setminus cs
\end{aligned}$$

provided

- $cs \cap vs = \varnothing$

Proof. Provided $cs \cap vs = \varnothing$, then

$$\begin{aligned}
&\left((P \setminus cs) \,/\!\!/_{vs}\, Rep_{MCell} \right)\\
&= \left(\begin{array}{l} P \setminus cs \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \setminus cs \end{array} \right) \setminus vs
\end{aligned}$$

[$cs \cap vs = \varnothing$ and [7, Section 3.5.1, L12]]

$$= \left(\left(\begin{array}{l} P \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \end{array} \right) \setminus cs \right) \setminus vs$$

[Roscoe [71, ⟨hide- ∥ -dist⟩(5.8)]]

$$= \left(\begin{array}{l} P \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(x_i)) \right) \end{array} \right) \setminus (vs \cup cs)$$

[Roscoe [71, ⟨hide-combine⟩(5.3)]]

$$= \left(\left(\left(\begin{array}{l} P \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\Big\|} \; i : 1..m \bullet (MCell_i\,(x_i)) \right) \end{array} \right) \setminus vs \right) \setminus cs$$

[Roscoe [71, ⟨hide-combine⟩(5.3)]]

$$= (\ P \mathbin{/\!\!/}_{vs} Rep_{MCell}\ ) \setminus cs$$ [Abbreviations C.1.1]

□

C.1.1.17 Hiding and Sequential composition - distribution

Law C.1.16 (Hiding and Sequential composition - distribution).

$$(CP \setminus cs)\,;\, end \rightarrow SKIP = (CP\,;\, end \rightarrow SKIP) \setminus cs$$

provided

$$end \notin cs \land end \notin usedC\,(CP)$$

Proof. The proof of this law is very similar to that of Law C.1.6.

(1) If CP is $STOP$, then the LHS of the law

$(CP \setminus cs)\,;\, end \rightarrow SKIP$

$= (STOP \setminus cs)\,;\, end \rightarrow SKIP$

$= STOP\,;\, end \rightarrow SKIP$ [Hoare [7, Section 3.5.1, L4]]

$= STOP$ [Hoare [7, Section 5.2, L5]]

and the RHS of the law

$(CP\,;\, end \rightarrow SKIP) \setminus cs$

$= (STOP\,;\, end \rightarrow SKIP) \setminus cs$

$= (STOP) \setminus cs$ [Hoare [7, Section 5.2, L5]]

$= STOP$ [Hoare [7, Section 3.5.1, L4]]

therefore the law is valid.

(2) If CP is $SKIP$, then the LHS of the law

$(CP \setminus cs)\,;\, end \rightarrow SKIP$

$= (SKIP \setminus cs)\,;\, end \rightarrow SKIP$

$= SKIP\,;\, end \rightarrow SKIP$ [Roscoe [71, ⟨$SKIP$hide-Id⟩(13.1)]]

$= end \rightarrow SKIP$ [Roscoe [71, ⟨;-unit-l⟩(7.2)]]

and the RHS of the law

$(CP\,;\, end \rightarrow SKIP) \setminus cs$

$= (SKIP\,;\, end \rightarrow SKIP) \setminus cs$

$= (end \rightarrow SKIP) \setminus cs$ [Roscoe [71, ⟨;-unit-l⟩(7.2)]]

$= end \rightarrow SKIP$ [Roscoe [71, ⟨hide-step 1⟩(5.5)] and [71, ⟨$SKIP$hide-Id⟩(13.1)]]

therefore the law is valid.

(3) If CP is **div**, then the LHS of the law

$$\begin{aligned}
&(CP \setminus cs) \mathbin{;} end \to SKIP\\
&= (\mathbf{div} \setminus cs) \mathbin{;} end \to SKIP\\
&= \mathbf{div} \mathbin{;} end \to SKIP && \text{[Hoare [7, Section 3.8.1, L2]]}\\
&= \mathbf{div} && \text{[Hoare [7, Section 5.3.2, L1]]}
\end{aligned}$$

and the RHS of the law

$$\begin{aligned}
&(CP \mathbin{;} end \to SKIP) \setminus cs\\
&= (\mathbf{div} \mathbin{;} end \to SKIP) \setminus cs\\
&= \mathbf{div} \setminus cs && \text{[Hoare [7, Section 5.3.2, L1]]}\\
&= \mathbf{div} && \text{[Hoare [7, Section 3.8.1, L2]]}
\end{aligned}$$

therefore the law is valid.

(4) If CP is an internal choice $CP_1 \sqcap CP_2$, assume

$$(CP_1 \setminus cs) \mathbin{;} end \to SKIP = (CP_1 \mathbin{;} end \to SKIP) \setminus cs$$

and

$$(CP_2 \setminus cs) \mathbin{;} end \to SKIP = (CP_2 \mathbin{;} end \to SKIP) \setminus cs$$

then the LHS of the law

$$\begin{aligned}
&(CP \setminus cs) \mathbin{;} end \to SKIP\\
&= ((CP_1 \sqcap CP_2) \setminus cs) \mathbin{;} end \to SKIP\\
&= ((CP_1 \setminus cs) \sqcap (CP_2 \setminus cs)) \mathbin{;} end \to SKIP && \text{[Hoare [7, Section 3.5.1, L3]]}\\
&= ((CP_1 \setminus cs) \mathbin{;} end \to SKIP) \sqcap ((CP_2 \setminus cs) \mathbin{;} end \to SKIP)\\
& && \text{[Hoare [7, Section 5.3.2, L2A]]}
\end{aligned}$$

and the RHS of the law

$$\begin{aligned}
&(CP \mathbin{;} end \to SKIP) \setminus cs\\
&= ((CP_1 \sqcap CP_2) \mathbin{;} end \to SKIP) \setminus cs\\
&= ((CP_1 \mathbin{;} end \to SKIP) \sqcap (CP_2 \mathbin{;} end \to SKIP)) \setminus cs\\
& && \text{[Hoare [7, Section 5.3.2, L2A]]}\\
&= ((CP_1 \mathbin{;} end \to SKIP) \setminus cs) \sqcap ((CP_2 \mathbin{;} end \to SKIP) \setminus cs)\\
& && \text{[Hoare [7, Section 3.5.1, L3]]}\\
&= ((CP_1 \setminus cs) \mathbin{;} end \to SKIP) \sqcap ((CP_2 \setminus cs) \mathbin{;} end \to SKIP) && \text{[Assumption]}
\end{aligned}$$

therefore the law is valid for the situation that the process is internal choice. In other words, that the process in the hiding is an internal choice preserves the distribution of hiding and sequential composition in the memory model.

(4) If CP is a prefixing, provided $CP = ?x : A \to CP'$, and $A = A_o \cup A_{cs}$ where A_{cs} denotes a set of events from cs and A_o for other events, and assume

$$\big(CP' \setminus cs\big) \mathbin{;} end \to SKIP = \big(CP' \mathbin{;} end \to SKIP\big) \setminus cs$$

then the LHS of the law

$$(CP \setminus cs) \mathbin{;} end \to SKIP$$

$$= \begin{cases} (CP' \setminus cs) \,;\, end \to SKIP & \text{if } x \in A_{cs} \\ ?x : A_o \to (CP' \setminus cs) \,;\, end \to SKIP & \text{if } x \in A_o \end{cases}$$

[Roscoe [71, ⟨hide-step 1⟩(5.5)]]

$$= \begin{cases} ((CP' \setminus cs) \,;\, end \to SKIP) & \text{if } x \in A_{cs} \\ ?x : A_o \to ((CP' \setminus cs) \,;\, end \to SKIP) & \text{if } x \in A_o \end{cases}$$

[Roscoe [71, ⟨;-step⟩(7.4)]]

And the RHS of the law

$$(CP \,;\, end \to SKIP) \setminus cs$$

$$= \big(?x : A \to \big(CP' \,;\, end \to SKIP\big)\big) \setminus cs \qquad \text{[Roscoe [71, ⟨;-step⟩(7.4)]]}$$

$$= \begin{cases} (CP' \,;\, end \to SKIP) \setminus cs & \text{if } x \in A_{cs} \\ ?x : A_o \to ((CP' \,;\, end \to SKIP) \setminus cs) & \text{if } x \in A_o \end{cases}$$

[Roscoe [71, ⟨hide-step 1⟩(5.5)]]

$$= \begin{cases} ((CP' \setminus cs) \,;\, end \to SKIP) & \text{if } x \in A_{cs} \\ ?x : A_o \to ((CP' \setminus cs) \,;\, end \to SKIP) & \text{if } x \in A_o \end{cases} \qquad \text{[Assumption]}$$

therefore the law is valid for the situation that the process is prefixing. In other words, that the process in the hiding is a prefixing preserves the distribution of hiding and sequential composition in the memory model.

Since this law is valid for all combinations of abstracted *CP* processes, this law is valid for all *CP* processes. □

C.1.1.18 Parallel composition (disjoint variables) and Sequential composition - distribution

Law C.1.17 (Parallel composition (disjoint variables) and Sequential composition - distribution).

$$\left(\left(\left(CP_1 \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \right)$$

$$= \left(\begin{array}{l} (\ (CP_1 \,;\, end \to SKIP) /\!\!/_{vs_1}\ Rep_{MCell_1}\ ) \\ \underset{cs}{\parallel} \\ (\ (CP_2 \,;\, end \to SKIP) /\!\!/_{vs_2}\ Rep_{MCell_2}\ ) \end{array} \right)$$

provided

$$cs \cap vs = \varnothing$$

$$scpV\,(CP_1) \cap scpV\,(CP_2) = \varnothing$$

$$l_{CP_1} = lclV\,(CP_1) = \{l_1, \ldots, l_p\}$$

$$l_{CP_2} = lclV\,(CP_2) = \{l_{p+1}, \ldots, l_m\}$$

$$vs_1 \cup vs_2 = vs$$

$$Rep_{MCell_1} = \left(\underset{\{\!|end|\!\}}{\parallel}\ i : 1..p \bullet (MCell_i\,(x_i)) \right)$$

$$Rep_{MCell_2} = \left(\underset{\{\!|end|\!\}}{\parallel}\ i : (p+1)..m \bullet (MCell_i\,(x_i)) \right)$$

Proof. The proof of this law is very similar to that of Law C.1.6.

(1) If both CP_1 and CP_2 are $STOP$, it can be easily proved that both LHS and RHS will result in $STOP$ too. Therefore, the law is valid.

(2) If CP_1 is **div**, it can be easily proved that both LHS and RHS will result in **div** too. Therefore, the law is valid.

(3) If both CP_1 and CP_2 are $SKIP$, it can be easily proved that both LHS and RHS will result in $SKIP$ too. Therefore, the law is valid.

(4) If CP_1 is an internal choice $CP_{11} \sqcap CP_{12}$, and assume

$$\left(\left(\left(CP_{11} \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \right) = \begin{pmatrix} (\ (CP_{11} ; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell_1}\ ) \\ \underset{cs}{\parallel} \\ (\ (CP_2 ; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell_2}\ ) \end{pmatrix}$$

and

$$\left(\left(\left(CP_{12} \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \right) = \begin{pmatrix} (\ (CP_{12} ; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell_1}\ ) \\ \underset{cs}{\parallel} \\ (\ (CP_2 ; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell_2}\ ) \end{pmatrix}$$

then the LHS of the law

$$\begin{aligned}
&\left(\left(\left(CP_1 \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \right) \\
&= \left(\left(\left((CP_{11} \sqcap CP_{12}) \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \right) \\
&= \left(\left(\begin{pmatrix} \left(CP_{11} \underset{cs}{\parallel} CP_2 \right) \\ \sqcap \\ \left(CP_{12} \underset{cs}{\parallel} CP_2 \right) \end{pmatrix} ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \right) && \text{[Hoare [7, Section 3.2.1, L7]]} \\
&= \begin{pmatrix} \left(\left(CP_{11} \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \\ \sqcap \\ \left(\left(CP_{12} \underset{cs}{\parallel} CP_2 \right) ; end \to SKIP \right) /\!\!/_{vs}\ Rep_{MCell} \end{pmatrix} && \text{[Law C.1.10]}
\end{aligned}$$

and the RHS of the law

$$\begin{aligned}
&\begin{pmatrix} (\ (CP_1 ; end \to SKIP) /\!\!/_{vs_1}\ Rep_{MCell_1}\ ) \\ \underset{cs}{\parallel} \\ (\ (CP_2 ; end \to SKIP) /\!\!/_{vs_2}\ Rep_{MCell_2}\ ) \end{pmatrix} \\
&= \begin{pmatrix} (\ ((CP_{11} \sqcap CP_{12}) ; end \to SKIP) /\!\!/_{vs_1}\ Rep_{MCell_1}\ ) \\ \underset{cs}{\parallel} \\ (\ (CP_2 ; end \to SKIP) /\!\!/_{vs_2}\ Rep_{MCell_2}\ ) \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} (CP_{11} ; end \to SKIP) /\!\!/_{vs_1}\ Rep_{MCell_1} \\ \sqcap \\ (CP_{12} ; end \to SKIP) /\!\!/_{vs_1}\ Rep_{MCell_1} \end{pmatrix} \\ \underset{cs}{\parallel}\ (\ (CP_2 ; end \to SKIP) /\!\!/_{vs_2}\ Rep_{MCell_2}\ ) \end{pmatrix} && \text{[Law C.1.10]}
\end{aligned}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} (CP_{11} \,;\, end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1} \\ \underset{cs}{\parallel} \\ (CP_2 \,;\, end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2} \end{array}\right) \\ \sqcap \\ \left(\begin{array}{l} (CP_{12} \,;\, end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1} \\ \underset{cs}{\parallel} \\ (CP_2 \,;\, end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2} \end{array}\right) \end{array}\right)$$

[Hoare [7, Section 3.2.1, L7]]

$$= \left(\begin{array}{l} \left(\left(CP_{11} \underset{cs}{\parallel} CP_2\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \\ \sqcap \\ \left(\left(CP_{12} \underset{cs}{\parallel} CP_2\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \end{array}\right)$$

[Assumption]

therefore, the law is valid for the situation that one of processes is internal choice. In other words, that one of process in the external choice is an internal choice preserves the distribution of parallel composition and sequential composition in the memory model.

(4) If CP_1 is a prefixing, and CP_2 is a prefixing as well, assume $CP_1 = ?e : A \to CP_1'$ and $CP_2 = ?e : B \to CP_2'$ where A ($A = A_{vs} \cup A_{cs} \cup A_o$, in which A_{vs} for events in vs, A_{cs} for evens in cs, and A_o for independent events) and B ($B = B_{vs} \cup B_{cs} \cup B_o$) denote a set of initially available events for CP_1 and CP_2 respectively. Since CP_1 and CP_2 have disjoint variables in scope, $A_{vs} \cap B_{vs} = \varnothing$.

The LHS of the law

$$\left(\left(\left(CP_1 \underset{cs}{\parallel} CP_2\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right)$$

$$= \left(\left(\left((?e : A \to CP_1') \underset{cs}{\parallel} (?e : B \to CP_2')\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right)$$

$$= \left(\left(\begin{array}{l} \left(\begin{array}{l} ?e : A_{vs} \to \left(CP_1' \underset{cs}{\parallel} CP_2\right) \\ \Box ?e : A_o \to \left(CP_1' \underset{cs}{\parallel} CP_2\right) \\ \Box ?e : (A_{cs} \cap B_{cs}) \to \left(CP_1' \underset{cs}{\parallel} CP_2'\right) \\ \Box ?e : B_{vs} \to \left(CP_1 \underset{cs}{\parallel} CP_2'\right) \\ \Box ?e : B_o \to \left(CP_1 \underset{cs}{\parallel} CP_2'\right) \end{array}\right) \\ ; end \to SKIP \end{array} \right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right)$$

[Roscoe [71, $\langle \parallel$ -step$\rangle$(3.10)]]

$$= \left(\begin{array}{l} \left(\left(\left(CP_1' \underset{cs}{\parallel} CP_2\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right) \\ \Box ?e : A_o \to \left(\left(\left(CP_1' \underset{cs}{\parallel} CP_2\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right) \\ \Box ?e : (A_{cs} \cap B_{cs}) \to \left(\left(\left(CP_1' \underset{cs}{\parallel} CP_2'\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right) \\ \Box ?e : B_o \to \left(\left(\left(CP_1 \underset{cs}{\parallel} CP_2'\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right) \\ \Box \left(\left(\left(CP_1 \underset{cs}{\parallel} CP_2'\right) ; end \to SKIP\right) \mathbin{/\!\!/}_{vs} Rep_{MCell} \right) \end{array}\right)$$

[Law C.1.8]

and the RHS of the law

$$
\begin{aligned}
&\left(\begin{array}{l}
\left(\ (CP_1\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\ \right)\\
\underset{cs}{\|}\\
\left(\ (CP_2\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}\ \right)
\end{array}\right)\\
=&\left(\begin{array}{l}
\left(\ (?e : A \to (CP_1'\,;\,end \to SKIP)) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\ \right)\\
\underset{cs}{\|}\\
\left(\ (?e : B \to (CP_2'\,;\,end \to SKIP)) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}\ \right)
\end{array}\right)\\
&\hfill\text{[Roscoe [71, }\langle;\text{-step}\rangle(7.4)]]\\
=&\left(\begin{array}{l}
\left(\begin{array}{l}
(CP_1'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\Box\\
?e : (A_o \cup A_{cs}) \to ((CP_1'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1})
\end{array}\right)\\
\underset{cs}{\|}\\
\left(\begin{array}{l}
(CP_2'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\Box\\
?e : (B_o \cup B_{cs}) \to ((CP_2'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1})
\end{array}\right)
\end{array}\right)\\
&\hfill\text{[Roscoe [71, }\langle\|\text{ -step}\rangle(3.10)]]\\
=&\left(\begin{array}{l}
\left(\begin{array}{l}
(CP_1'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\underset{cs}{\|}\\
(CP_2\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}
\end{array}\right)\\
\Box ?e : A_o \to \left(\begin{array}{l}
(CP_1'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\underset{cs}{\|}\\
(CP_2\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}
\end{array}\right)\\
\Box ?e : (A_{cs} \cap B_{cs}) \to \left(\begin{array}{l}
(CP_1'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\underset{cs}{\|}\\
(CP_2'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}
\end{array}\right)\\
\Box ?e : B_o \to \left(\begin{array}{l}
(CP_1\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\underset{cs}{\|}\\
(CP_2'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}
\end{array}\right)\\
\Box \left(\begin{array}{l}
(CP_1\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\\
\underset{cs}{\|}\\
(CP_2'\,;\,end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}
\end{array}\right)
\end{array}\right)\\
&\hfill\text{[Roscoe [71, }\langle\|\text{ -step}\rangle(3.10)]]
\end{aligned}
$$

The parallel composition is rewritten to external choice and prefixing by its step rule. The one-step application of the rule for both LHS and RHS is shown above. And we can find that both LHS and RHS result in the external choices having the same initial events. The same result can be got by further applications of the step rule. Therefore the law is valid for the situation that both processes are prefixing.

(5) In addition, if $CP_1 \underset{cs}{\|} CP_2$ terminates, then both CP_1 and CP_2 will terminate, and the LHS terminates according Lemma C.1.2. And the RHS will terminate as well because both the memory model of CP_1 and the memory model of CP_2 terminate according to Lemma C.1.2. If $P \underset{cs}{\|} Q$ does not terminate, then at least one of P and Q does not terminate. Thus both LHS and RHS will not terminate.

Since this law is valid for all combinations of abstracted CP processes, this law is valid for all CP processes.

$\Box$

C.1.1.19 Interleaving (disjoint variables) and Sequential composition - distribution

Law C.1.18 (Interleaving (disjoint variables) and Sequential composition - distribution).

$$\begin{array}{l} \left(\ ((CP_1 \mid\mid\mid CP_2)\ ; end \to SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell}\ \right) \\ = \left(\begin{array}{l} \left(\ (CP_1\ ; end \to SKIP) \mathbin{/\!\!/}_{vs_1} Rep_{MCell_1}\ \right) \\ \mid\mid\mid \\ \left(\ (CP_2\ ; end \to SKIP) \mathbin{/\!\!/}_{vs_2} Rep_{MCell_2}\ \right) \end{array} \right) \end{array}$$

provided

$$cs \cap vs = \varnothing$$

$$scpV\,(CP_1) \cap scpV\,(CP_2) = \varnothing$$

$$l_{CP_1} = lclV\,(CP_1) = \{l_1, \ldots, l_p\}$$

$$l_{CP_2} = lclV\,(CP_2) = \{l_{p+1}, \ldots, l_m\}$$

$$vs_1 \cup vs_2 = vs$$

$$Rep_{MCell_1} = \left(\mathop{\Vert}_{\{\!|end|\!\}} i : 1..p \bullet (MCell_i\,(x_i)) \right)$$

$$Rep_{MCell_2} = \left(\mathop{\Vert}_{\{\!|end|\!\}} i : (p+1)..m \bullet (MCell_i\,(x_i)) \right)$$

Proof. It is omitted for brevity because its proof is very similar to the proof of Law C.1.17. □

C.1.1.20 Memory model - combination

Law C.1.19 (Memory model - combination).

$$\begin{array}{ll} & F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet F_{Mem}\,(CP, \{y\}) \right), \{x\} \right) \\ = & \bigsqcap_{y:Ty} \bullet F_{Mem}\,(CP, \{x, y\}) \end{array}$$

Proof. (1) For a basic process CP ($STOP$, $SKIP$, or **div**), the LHS of the law

$$\begin{array}{lll} & F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet F_{Mem}\,(CP, \{y\}) \right), \{x\} \right) & \\ = & F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet CP \right), \{x\} \right) & [F_{Var} \text{ Rule 2, and Definition B.1.5}] \\ = & F_{Mem}\,(CP, \{x\}) & [\text{Roscoe [71, } \langle \sqcap\text{-idem} \rangle (2.2)]] \\ = & CP & [F_{Var} \text{ Rule 2, and Definition B.1.5}] \end{array}$$

And the RHS of the law

$$\begin{array}{lll} & \bigsqcap_{y:Ty} \bullet F_{Mem}\,(CP, \{x, y\}) & \\ = & \bigsqcap_{y:Ty} \bullet CP & [F_{Var} \text{ Rule 2, and Definition B.1.5}] \\ = & CP & [\text{Roscoe [71, } \langle \sqcap\text{-idem} \rangle (2.2)]] \end{array}$$

Therefore, for basic processes the law is proved.

(2) If CP is an internal choice $CP_1 \sqcap CP_2$, since internal choice distributes through $\bigsqcap$ and the memory model (Law C.1.10), we can easily prove the law is valid. Finally, its proof is omitted for brevity.

(3) If CP is a prefixing, and assume

$$F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet F_{Mem}(CP, \{y\})\right), \{x\}\right) = \bigsqcap_{y:Ty} \bullet F_{Mem}(CP, \{x, y\})$$

then we need to prove

$$\begin{aligned} & F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet F_{Mem}(c!e(x,y)?x?y \to CP, \{y\})\right), \{x\}\right) \\ = \; & \bigsqcap_{y:Ty} \bullet F_{Mem}(c!e(x,y)?x?y \to CP, \{x, y\}) \end{aligned}$$

where the communication $c!e(x,y)?x?y$ is an abstraction that means both x and y are evaluated and updated in the communication. Then

$$F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet F_{Mem}(c!e(x,y)?x?y \to CP, \{y\})\right), \{x\}\right)$$

$$= F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l} \left(\begin{array}{l} set_y!y_i \to SKIP; \\ F_{Var}(c!e(x,y)?x?y \to CP, \{y\}) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs_y}{\parallel} MemCell_y \end{array}\right) \setminus vs_y\right), \{x\}\right)$$

[Definition B.1.5]

$$= F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l} \left(\begin{array}{l} F_{Var}(c!e(x,y)?x?y \to CP, \{y\}) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs_y}{\parallel} MCell_y(y_i) \end{array}\right) \setminus vs_y\right), \{x\}\right)$$

[Lemma C.1.1]

$$= F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l} \left(\begin{array}{l} get_y?y \to c!e(x,y)?x?y \to \\ set_y!y \to F_{Var}(CP, \{y\}) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs_y}{\parallel} MCell_y(y_i) \end{array}\right) \setminus vs_y\right), \{x\}\right)$$

[F_{Var} Rule 3]

$$= F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l} \left(\begin{array}{l} c!e(x,y_i)?x?y \to set_y!y \to \\ F_{Var}(CP, \{y\}) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs_y}{\parallel} MCell_y(y_i) \end{array}\right) \setminus vs_y\right), \{x\}\right)$$

[Law C.1.3]

$$= F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l} \left(\begin{array}{l} c!e(x,y_i)?x?y_j \to set_y!y_j \to \\ F_{Var}(CP, \{y\}) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs_y}{\parallel} MCell_y(y_i) \end{array}\right) \setminus vs_y\right), \{x\}\right)$$

[Replace bounded variable name where $y_j \in Ty$]

$$= F_{Mem}\left(\left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l} \left(\begin{array}{l} c!e(x,y_i)?x?y_j \to set_y!y_j \to \\ (F_{Var}(CP, \{y\}); end \to SKIP) \end{array}\right) \\ \underset{vs_y}{\parallel} MCell_y(y_i) \end{array}\right) \setminus vs_y\right), \{x\}\right)$$

[Roscoe [71, ⟨;-step⟩(7.4)]]

$$
= F_{Mem}\left(\left(\begin{array}{l}
\bigsqcap_{y:Ty} \bullet c!e\,(x, y_i)?x?y_j \to \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l} set_y!y_j \to \\ (F_{Var}\,(CP, \{y\})\,;\, end \to SKIP) \end{array}\right) \\
\underset{vs_y}{\parallel}\ MCell_y\,(y_i)
\end{array}\right) \setminus vs_y
\end{array}\right), \{x\}\right)
$$

[Roscoe [71, ⟨∥ -step⟩(3.10)] and hiding]

$$
= F_{Mem}\left(\left(\begin{array}{l}
\bigsqcap_{y:Ty} \bullet c!e\,(x, y_i)?x?y_j \to \\
\qquad \left(\begin{array}{l}
(F_{Var}\,(CP, \{y\})\,;\, end \to SKIP) \\
\underset{vs_y}{\parallel}\ MCell_y\,(y_j)
\end{array}\right) \setminus vs_y
\end{array}\right), \{x\}\right)
$$

[Law C.1.4]

$$
= \begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
set_x!x_i \to SKIP; \\
F_{Var}\left(\begin{array}{l}
\left(\begin{array}{l}
\bigsqcap_{y:Ty} \bullet c!e\,(x, y_i)?x?y_j \to \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \to SKIP \end{array}\right) \\
\underset{vs_y}{\parallel}\ MCell_y\,(y_j)
\end{array}\right) \setminus vs_y
\end{array}\right) \\
, \{x\}
\end{array}\right) \\
;end \to SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MemCell_x
\end{array}\right) \\
\setminus vs_x
\end{array}
$$

[Definition B.1.5]

$$
= \begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
F_{Var}\left(\begin{array}{l}
\left(\begin{array}{l}
\bigsqcap_{y:Ty} \bullet c!e\,(x, y_i)?x?y_j \to \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \to SKIP \end{array}\right) \\
\underset{vs_y}{\parallel}\ MCell_y\,(y_j)
\end{array}\right) \setminus vs_y
\end{array}\right) \\
, \{x\}
\end{array}\right) \\
;end \to SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_i)
\end{array}\right) \\
\setminus vs_x
\end{array}
$$

[Lemma C.1.1]

$$
= \begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
get_x?x \to \\
\bigsqcap_{y:Ty} \bullet c!e\,(x, y_i)?x?y_j \to set_x!x \to \\
\qquad F_{Var}\left(\left(\begin{array}{l}
\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \to SKIP \end{array}\right) \\
\underset{vs_y}{\parallel}\ MCell_y\,(y_j)
\end{array}\right) \setminus vs_y, \{x\}\right)
\end{array}\right) \\
;end \to SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_i)
\end{array}\right) \\
\setminus vs_x
\end{array}
$$

[F_{Var} Rule 14 and 3]

$$
= \begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\sqcap_{y:Ty} \bullet c!e\,(x_i, y_i)?x_j?y_j \rightarrow set_x!x_j \rightarrow \\
\qquad F_{Var}\left(\left(\left(\begin{array}{l}\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \rightarrow SKIP \end{array}\right) \\ \underset{vs_y}{\parallel}\ MCell_y\,(y_j)\end{array}\right) \setminus vs_y, \{x\}\right)\right)
\end{array}\right) \\
;end \rightarrow SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_i)
\end{array}\right) \\
\setminus vs_x
\end{array}
$$

[Law C.1.3 and replace bounded variable name where $x_j \in Tx$]

$$
= \begin{array}{l}
\left(\begin{array}{l}
\sqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
c!e\,(x_i, y_i)?x_j?y_j \rightarrow set_x!x_j \rightarrow \\
F_{Var}\left(\begin{array}{l}\left(\left(\begin{array}{l}\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \rightarrow SKIP \end{array}\right) \\ \underset{vs_y}{\parallel}\ MCell_y\,(y_j)\end{array}\right) \setminus vs_y\right) \\ , \{x\}\end{array}\right) \\
;end \rightarrow SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_i)
\end{array}\right)
\end{array}\right) \\
\setminus vs_x
\end{array}
$$

[Law C.1.10]

$$
= \sqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
c!e\,(x_i, y_i)?x_j?y_j \rightarrow set_x!x_j \rightarrow \\
F_{Var}\left(\begin{array}{l}\left(\left(\begin{array}{l}\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \rightarrow SKIP \end{array}\right) \\ \underset{vs_y}{\parallel}\ MCell_y\,(y_j)\end{array}\right) \setminus vs_y\right) \\ , \{x\}\end{array}\right) \\
;end \rightarrow SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_i)
\end{array}\right) \setminus vs_x
$$

[Roscoe [71, $\langle$hide-dist$\rangle$(5.1)]]

$$
= \begin{array}{l}
\sqcap_{y:Ty} \bullet c!e\,(x_i, y_i)?x_j?y_j \rightarrow \\
\quad \left(\left(\begin{array}{l}
\left(\begin{array}{l}
set_x!x_j \rightarrow \\
F_{Var}\left(\begin{array}{l}\left(\left(\begin{array}{l}\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \rightarrow SKIP \end{array}\right) \\ \underset{vs_y}{\parallel}\ MCell_y\,(y_j)\end{array}\right) \setminus vs_y\right) \\ , \{x\}\end{array}\right) \\
;end \rightarrow SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_i)
\end{array}\right) \setminus vs_x\right)
\end{array}
$$

[Roscoe [71, $\langle\parallel$ -step$\rangle$(3.10)]]

$$
= \begin{array}{l}
\sqcap_{y:Ty} \bullet c!e\,(x_i, y_i)?x_j?y_j \rightarrow \\
\quad \left(\left(\begin{array}{l}
\left(\begin{array}{l}
F_{Var}\left(\begin{array}{l}\left(\left(\begin{array}{l}\left(\begin{array}{l} F_{Var}\,(CP, \{y\})\,; \\ end \rightarrow SKIP \end{array}\right) \\ \underset{vs_y}{\parallel}\ MCell_y\,(y_j)\end{array}\right) \setminus vs_y\right) \\ , \{x\}\end{array}\right) \\
;end \rightarrow SKIP
\end{array}\right) \\
\underset{vs_x}{\parallel}\ MCell_x\,(x_j)
\end{array}\right) \setminus vs_x\right)
\end{array}
$$

[Law C.1.4]

And the RHS of the law to be proved

$$
\begin{array}{l}
\left(\bigsqcap_{y:Ty} \bullet F_{Mem}\left(c!e(x,y)?x?y \to CP, \{x,y\}\right)\right) \\
= \left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
set_x!x_i \to set_y!y_i \to SKIP; \\
F_{Var}\left(c!e(x,y)?x?y \to CP, \{x,y\}\right) \\
;end \to SKIP
\end{array}\right) \\
\underset{vs}{\|} \\
\left(\underset{\{\!|end|\!\}}{\|} \{MemCell_x, MemCell_y\}\right)
\end{array}\right) \setminus vs\right)
\end{array}
$$

[Definition B.1.5]

$$
= \left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
F_{Var}\left(c!e(x,y)?x?y \to CP, \{x,y\}\right) \\
;end \to SKIP
\end{array}\right) \\
\underset{vs}{\|} \\
\left(\underset{\{\!|end|\!\}}{\|} \{MCell_x\left(x_i\right), MCell_y\left(y_i\right)\}\right)
\end{array}\right) \setminus vs\right)
$$

[Lemma C.1.1]

$$
= \left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
get_x?x \to get_y?y \to c!e(x,y)?x?y \to \\
set_x!x \to set_y!y \to \\
F_{Var}\left(CP, \{x,y\}\right); end \to SKIP
\end{array}\right) \\
\underset{vs}{\|} \\
\left(\underset{\{\!|end|\!\}}{\|} \{MCell_x\left(x_i\right), MCell_y\left(y_i\right)\}\right)
\end{array}\right) \setminus vs\right)
$$

[F_{Var} Rule 3]

$$
= \left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
c!e(x_i,y_i)?x?y \to set_x!x \to set_y!y \to \\
F_{Var}\left(CP, \{x,y\}\right); end \to SKIP
\end{array}\right) \\
\underset{vs}{\|} \\
\left(\underset{\{\!|end|\!\}}{\|} \{MCell_x\left(x_i\right), MCell_y\left(y_i\right)\}\right)
\end{array}\right) \setminus vs\right)
$$

[Law C.1.3]

$$
= \left(\bigsqcap_{y:Ty} \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
c!e(x_i,y_i)?x_j?y_j \to set_x!x_j \to set_y!y_j \to \\
F_{Var}\left(CP, \{x,y\}\right); end \to SKIP
\end{array}\right) \\
\underset{vs}{\|} \\
\left(\underset{\{\!|end|\!\}}{\|} \{MCell_x\left(x_i\right), MCell_y\left(y_i\right)\}\right)
\end{array}\right) \setminus vs\right)
$$

[Replace bounded variable name where $x_j \in Tx \land y_j \in Ty$]

$$
= \left(\begin{array}{l}
\bigsqcap_{y:Ty} \bullet c!e(x_i,y_i)?x_j?y_j \to \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
set_x!x_j \to set_y!y_j \to \\
F_{Var}\left(CP, \{x,y\}\right); end \to SKIP
\end{array}\right) \\
\underset{vs}{\|} \\
\left(\underset{\{\!|end|\!\}}{\|} \{MCell_x\left(x_i\right), MCell_y\left(y_i\right)\}\right)
\end{array}\right) \setminus vs
\end{array}\right)
$$

[Roscoe [71, $\langle \|$ -step$\rangle$(3.10)]]

$$= \left(\begin{array}{l} \sqcap_{y:Ty} \bullet c!e(x_i, y_i)?x_j?y_j \rightarrow \\ \quad \left(\begin{array}{l} \left(F_{Var}\left(CP, \{x, y\}\right) ; end \rightarrow SKIP \right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} \left\{MCell_x\left(x_j\right), MCell_y\left(y_j\right)\right\} \right) \end{array} \right) \setminus vs \end{array} \right) \qquad \text{[Law C.1.4]}$$

Compared to the LHS, then we need to prove

$$\left(\left(\begin{array}{l} \left(\begin{array}{l} F_{Var} \left(\begin{array}{l} \left(\begin{array}{l} (F_{Var}\left(CP, \{y\}\right) ; end \rightarrow SKIP) \\ \underset{vs_y}{\parallel} MCell_y\left(y_j\right) \end{array} \right) \setminus vs_y \\ , \{x\} \end{array} \right) \\ ; end \rightarrow SKIP \end{array} \right) \\ \underset{vs_x}{\parallel} MCell_x\left(x_j\right) \end{array} \right) \setminus vs_x \right)$$

$$= \left(\left(\begin{array}{l} F_{Var}\left(CP, \{x, y\}\right) ; end \rightarrow SKIP \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} \left\{MCell_x\left(x_j\right), MCell_y\left(y_j\right)\right\} \right) \end{array} \right) \setminus vs \right)$$

where $vs = vs_x \cup vs_y$.

By the similar rules, the equation can be easily derived from the assumption

$$F_{Mem}\left(\left(\sqcap_{y:Ty} \bullet F_{Mem}\left(CP, \{y\}\right) \right), \{x\} \right) \;=\; \sqcap_{y:Ty} \bullet F_{Mem}\left(CP, \{x, y\}\right)$$

Finally, the law is valid if CP is a prefixing.

Since this law is valid for all combinations of abstracted CP processes, this law is valid for all CP processes. $\square$

C.1.1.21 Memory model - termination

Law C.1.20 (Memory model - termination).

$$(end \rightarrow SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell} = SKIP$$

Proof.

$$(end \rightarrow SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell}$$

$$= \left((end \rightarrow SKIP) \underset{vs}{\parallel} \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(MCell_i\left(l_i\right)\right) \right) \right) \setminus vs$$

[Abbreviations C.1.1]

$$= \left(\begin{array}{l} (end \rightarrow SKIP) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(\begin{array}{ll} & set_i?y \rightarrow MCell_i\left(y\right) \\ \square & get_i!l_i \rightarrow MCell_i\left(l_i\right) \\ \square & end \rightarrow SKIP \end{array} \right) \right) \end{array} \right) \setminus vs$$

[Definition B.1.1]

$$= \left(SKIP \underset{vs}{\parallel} \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet (SKIP) \right) \right) \setminus vs \qquad \text{[Roscoe [71, } \langle \parallel \text{-step} \rangle (3.10)]]$$

$$= (SKIP) \setminus vs \qquad \text{[Roscoe [71, Distributed Termination]]}$$

$$= SKIP \qquad \text{[Roscoe [8, } \langle SKIP\text{-hide-Id} \rangle (6.9)]]$$

□

C.1.2 Lemmas

C.1.2.1 Memory Initialisation

Lemma C.1.1 (Memory Initialisation). *A set of MemCell processes for m local variables are initialised by a sequence of set events ($set_1!l_1, \cdots, set_m!l_m$).*

$$
\begin{aligned}
&(set_1!l_1 \rightarrow \cdots \rightarrow set_m!l_m \rightarrow SKIP \,;\, P) \parallel_{vs} Rep_{Mem} \\
&= \left(P \parallel_{vs} \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(l_i)) \right) \right)
\end{aligned}
$$

The states of all memory processes have been switched from MemCell to MCell, and the value of each l_i in its corresponding memory process $MCell_i$ is the message l_i from the set_i event.

Proof.

$$
\begin{aligned}
&(set_1!l_1 \rightarrow \cdots \rightarrow set_m!l_m \rightarrow SKIP \,;\, P) \parallel_{vs} Rep_{Mem} \\
&= \left(\begin{array}{l} (set_1!l_1 \rightarrow \cdots \rightarrow set_m!l_m \rightarrow SKIP \,;\, P) \\ \parallel_{vs} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MemCell_i) \right) \end{array} \right) && \text{[Abbreviations C.1.1]} \\
&= \left(\begin{array}{l} (set_2!l_2 \rightarrow \cdots \rightarrow set_m!l_m \rightarrow SKIP \,;\, P) \\ \parallel_{vs} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet \begin{cases} MemCell_i\,(x_i) & i \neq 1 \\ MCell_1\,(l_1) & i = 1 \end{cases} \right) \end{array} \right) && \text{[Law C.1.4]} \\
&= \left(\begin{array}{l} (set_3!l_3 \rightarrow \cdots \rightarrow set_m!l_m \rightarrow SKIP \,;\, P) \\ \parallel_{vs} \\ \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet \begin{cases} MemCell_i\,(x_i) & i \neq 1 \land i \neq 2 \\ MCell_1\,(l_1) & i = 1 \\ MCell_2\,(l_2) & i = 2 \end{cases} \right) \end{array} \right) && \text{[Law C.1.4]} \\
&= \left((SKIP \,;\, P) \parallel_{vs} \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(l_i)) \right) \right) \\
& && \text{[Applied Law C.1.4 } n \text{ times]} \\
&= \left(P \parallel_{vs} \left(\underset{\{\!|end|\!\}}{\parallel} i : 1..m \bullet (MCell_i\,(l_i)) \right) \right) && \text{[Roscoe [71, } \langle;\text{-unit-l}\rangle (7.2)]]
\end{aligned}
$$

□

C.1.2.2 Memory Model of Linked Actions

In this section, we aim to prove that the memory model of a CSP process $\Phi\,(R_{wrt}\,(A))$, that are linked from an action A in *Circus* by the link rule defined in our solution, preserves the same state and behaviour as the original action A. To prove the conclusion applied to each action, we assume that

- before the action and the memory model, the state variables and local variables in *Circus* have the same values as their counterparts (state variables in Z and memory processes in CSP) in the linked $CSP \parallel_B Z$,

and then to prove

- after the action and its linked memory model process in $CSP \parallel_B Z$, they get the same state and local variables as well as the same behaviour.

Our strategies are listed below.

- For an action A, we construct the memory model of its linked process in CSP by its corresponding link rule in Appendix E,

$$\left(\begin{array}{l} (Us \,;\, F_{Var}\,(\Phi\,(R_{wrt}\,(A)), l) \,;\, end \rightarrow SKIP) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} \; i : 1..m \bullet (set_i?x \rightarrow MCell_i\,(x)) \right) \end{array} \right) \setminus vs \qquad \text{[Definition B.1.5]}$$

- Then the memory model is transformed to a format that is syntactically similar to the linked process. For example, if the linked process is like $P \,;\, Q$, then the memory model is transformed to a sequential composition ; of the memory model of P and the memory model of Q. Since they have the same syntax and both are in CSP, we conclude they have the same behaviour.

- After that, we need to prove that the transformed process has the same state part as the original action A. That is, they have the same effect on state and local variables, where
  - state variables in *Circus* are within a basic process which the action belongs to, and state variables in $CSP \parallel_B Z$ are a part of global state space, (if the action does not update state variables, this step is omitted)
  - local variables in *Circus* are declared in variable blocks, and local variables in CSP actually denote their corresponding memory processes.

- Finally, if both the state and behavioural parts are kept, then we conclude the memory model of the linked process has the same semantics as the original action.

Furthermore, for the state variables evaluated in the first construct of an action, according to our link rules, R_{wrt} Rules will prefix the action with a number of $P\_OP\_s_i$ schema expressions to get the values of these state variables. Then they are translated to prefixing in CSP. Link Rule 39 is an example. These prefixing events are not a part of *get* and *set* events in the memory model. Therefore, they are regarded as normal events. To simplify the proof, we skip these additional events.

Termination of Memory Model

Lemma C.1.2 (Termination of Memory Model)**.** *The memory model of a process P terminates if and only if P terminates and is not a recursion.*

Proof. The memory model of a process P is

$$\begin{aligned} &((Us \,;\, F_{Var}\,(P, v) \,;\, end \rightarrow SKIP) \mathbin{/\!/}_{vs} Rep_{Mem}) && \text{[Definition B.1.5]} \\ &= (F_{Var}\,(P, v) \,;\, end \rightarrow SKIP) \mathbin{/\!/}_{vs} Rep_{MCell} && \text{[Lemma C.1.1]} \end{aligned}$$

Now considering F_{Var} rules, we tend to prove for each rule, if P terminates, $F_{Var}(P, v)$ also terminates

(1) If $scpV(P) \cap v = \varnothing$, then

$$\begin{aligned} &(F_{Var}(P, v) \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell} \\ &= (P \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell} && [F_{Var} \text{ Rule 1}] \end{aligned}$$

Since P terminates, according to Law C.1.20, then the process above also terminates

(2) If P is the basic process $SKIP$,

$$\begin{aligned} &(F_{Var}(SKIP, v) \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell} \\ &= (SKIP \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell} && [F_{Var} \text{ Rule 2}] \\ &= (end \to SKIP) \parallel_{vs} Rep_{MCell} && [\text{Roscoe [71, } \langle\text{;-unit-l}\rangle(7.2)]] \\ &= SKIP && [\text{Law C.1.20}] \end{aligned}$$

there the memory model of P also terminates.

(3) If P is a prefixing like $c!e\,(l_p, \cdots, l_q)?l_i \to P'$, assume the memory model of P' terminates, then

$$\begin{aligned} &\left(F_{Var}\left(c!e\,(l_p, \cdots, l_q)?l_i \to P', v\right) ; end \to SKIP\right) \parallel_{vs} Rep_{MCell} \\ &= \begin{pmatrix} get_p?l_p \to \cdots \to get_q?l_q \to c!e\,(l_p, \cdots, l_q)?l_i \to set_i!l_i \to \\ F_{Var}(P', v) \,;\, end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \\ & && [F_{Var} \text{ Rule 3}] \\ &= \begin{pmatrix} c!e\,(x_p, \cdots, x_q)?l_i \to set_i!l_i \to \\ F_{Var}(P', v) \,;\, end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} && [\text{Law C.1.3}] \\ &= c!e\,(x_p, \cdots, x_q)?l_i \to \big(\big(\ set_i!l_i \to F_{Var}(P', v) \,;\, end \to SKIP\ \big) \parallel_{vs} Rep_{MCell}\big) \\ & && [\text{Hoare [7, Section 4.5.1, L2]}] \\ &= c!e\,(x_p, \cdots, x_q)?l_i \to \big(\big(\ F_{Var}(P', v) \,;\, end \to SKIP\ \big) \parallel_{vs} Rep_{MCell}\big) \\ & && [\text{Law C.1.4}] \end{aligned}$$

According to assumption, the memory model of this prefixing also terminates. In other words, prefixing $c \to P$ will not cause non-termination if P terminates.

(4) If P is a sequential composition like $P_1 \,;\, P_2$, assume

$$initRdLclV(P_1) = \{l_p, \cdots, l_q\}$$

and the memory models of both P_1 and P_2 terminate, then

$$\begin{aligned} &(F_{Var}(P_1 \,;\, P_2, v) \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell} \\ &= \begin{pmatrix} get_p?l_p \to \cdots \to get_q?l_q \to (F_{VarPost}(P_1, v) \,;\, F_{Var}(P_2, v)) \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \\ & && [F_{Var} \text{ Rule 4}] \\ &= \begin{pmatrix} (F_{VarPost}(P_1[\![x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}]\!], v) \,;\, F_{Var}(P_2, v)) \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \\ & && [\text{Law C.1.3}] \\ &= \begin{pmatrix} F_{VarPost}(P_1[\![x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}]\!], v) \,; \\ (F_{Var}(P_2, v) \,;\, end \to SKIP) \end{pmatrix} \parallel_{vs} Rep_{MCell} \\ & && [\text{Hoare [7, Section 5.2, L2]}] \\ &= \begin{pmatrix} (F_{VarPost}(P_1[\![x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}]\!], v) \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell_1} \\ ;(F_{Var}(P_2, v) \,;\, end \to SKIP) \parallel_{vs} Rep_{MCell_2} \end{pmatrix} \\ & && [\text{Law C.1.14}] \end{aligned}$$

where $Rep_{MCell_1} = Rep_{MCell} \neq Rep_{MCell_2}$, and Rep_{MCell_2} is very similar to Rep_{MCell_1} except that some local variables are updated by P_1.

According to assumption, the memory models of both P_1 and P_2 terminate. That is, the process before ; terminates and the process after ; terminates. And finally the sequential composition terminates. In other words, the memory model of sequential composition $P_1 \, ; P_2$ will not cause non-termination if the memory models of P_1 and P_2 terminate.

(5) If P is a process hiding like $P \setminus cs$, assume

$$\begin{aligned} &initRdLclV\,(P) = \{l_p, \cdots, l_q\} \\ &cs \cap vs = \varnothing \end{aligned}$$

and the memory model of P terminates, then

$$\begin{aligned}
&(F_{Var}\,(P \setminus cs, v) \, ; end \to SKIP) \parallel_{vs} Rep_{MCell} \\
&= \begin{pmatrix} get_p?l_p \to \cdots \to get_q?l_q \to (F_{VarPost}\,(P, v) \setminus cs) \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \\
&\hfill [F_{Var} \text{ Rule 5}] \\
&= \begin{pmatrix} (F_{VarPost}\,(P[\![x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}]\!], v) \setminus cs) \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \qquad [\text{Law C.1.3}] \\
&= \left(\begin{pmatrix} F_{VarPost}\,(P[\![x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}]\!], v) \\ ;end \to SKIP \end{pmatrix} \setminus cs \right) \parallel_{vs} Rep_{MCell} \\
&\hfill [\text{Law C.1.16}] \\
&= \left(\begin{pmatrix} F_{VarPost}\,(P[\![x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}]\!], v) \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \right) \setminus cs \\
&\hfill [\text{Law C.1.15}]
\end{aligned}$$

According to assumption, the memory model of P terminates, then the hiding of cs from it terminates as well. In other words, the memory model of process hiding $P \setminus cs$ will not cause non-termination if the memory model of P terminates.

(6) If P is an external choice like $P_1 \Box P_2$, assume

$$\begin{aligned} &initRdLclV\,(P_1) = \{l_{p1}, \cdots, l_{q1}\} \\ &initRdLclV\,(P_2) = \{l_{p2}, \cdots, l_{q2}\} \\ &initRdLclV\,(P_1) \cup initRdLclV\,(P_2) = \{l_p, \cdots, l_q\} \end{aligned}$$

and the memory models of both P_1 and P_2 terminate, then

$$\begin{aligned}
&(F_{Var}\,(P_1 \Box P_2, v) \, ; end \to SKIP) \parallel_{vs} Rep_{MCell} \\
&= \begin{pmatrix} get_p?l_p \to \cdots \to get_q?l_q \to (F_{VarPost}\,(P_1, v) \Box F_{VarPost}\,(P_2, v)) \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \\
&\hfill [F_{Var} \text{ Rule 6}] \\
&= \begin{pmatrix} \begin{pmatrix} F_{VarPost}\,(P_1[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!], v) \\ \Box\, F_{VarPost}\,(P_2[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!], v) \end{pmatrix} \\ ;end \to SKIP \end{pmatrix} \parallel_{vs} Rep_{MCell} \\
&\hfill [\text{Law C.1.3}] \\
&= \begin{pmatrix} (F_{VarPost}\,(P_1[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!], v) \, ; end \to SKIP) \parallel_{vs} Rep_{MCell} \\ \Box \\ (F_{VarPost}\,(P_2[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!], v) \, ; end \to SKIP) \parallel_{vs} Rep_{MCell} \end{pmatrix} \\
&\hfill [\text{Law C.1.8}]
\end{aligned}$$

According to assumption, the memory models of both P_1 and P_2 terminate. Therefore, the external choice of them also terminates. In other words, the memory model of external choice $P_1 \Box P_2$ will not cause non-termination if the memory models of P_1 and P_2 terminate.

(7) If P is an internal choice like $P_1 \sqcap P_2$, assume

$$\begin{aligned}
&initRdLclV\,(P_1) = \{l_{p1}, \cdots, l_{q1}\}\\
&initRdLclV\,(P_2) = \{l_{p2}, \cdots, l_{q2}\}\\
&initRdLclV\,(P_1) \cup initRdLclV\,(P_2) = \{l_p, \cdots, l_q\}
\end{aligned}$$

and the memory models of both P_1 and P_2 terminate, then

$$\begin{aligned}
&(F_{Var}\,(P_1 \sqcap P_2, v)\,;\, end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell}\\
&= \left(\begin{array}{l} get_p?l_p \to \cdots \to get_q?l_q \to (F_{VarPost}\,(P_1, v) \sqcap F_{VarPost}\,(P_2, v))\\ ;end \to SKIP \end{array}\right) \mathbin{/\!\!/_{vs}} Rep_{MCell} && [F_{Var} \text{ Rule 7}]\\
&= \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}\,(P_1[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!], v)\\ \sqcap F_{VarPost}\,(P_2[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!], v) \end{array}\right)\\ ;end \to SKIP \end{array}\right) \mathbin{/\!\!/_{vs}} Rep_{MCell} && [\text{Law C.1.3}]\\
&= \left(\begin{array}{l} (F_{VarPost}\,(P_1[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!], v)\,;\, end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell}\\ \sqcap\\ (F_{VarPost}\,(P_2[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!], v)\,;\, end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell} \end{array}\right) && [\text{Law C.1.10}]
\end{aligned}$$

According to assumption, the memory models of both P_1 and P_2 terminate. Therefore, the internal choice of them also terminates. In other words, the memory model of internal choice $P_1 \sqcap P_2$ will not cause non-termination if the memory models of P_1 and P_2 terminate.

(8) If P is a boolean guard like $b \;\&\; P$, assume

$$\begin{aligned}
&lclV\,(b) = \{l_{p1}, \cdots, l_{q1}\}\\
&initRdLclV\,(P) = \{l_{p2}, \cdots, l_{q2}\}\\
&lclV\,(b) \cup initRdLclV\,(P) = \{l_p, \cdots, l_q\}
\end{aligned}$$

and the memory model of P terminates, then

$$\begin{aligned}
&(F_{Var}\,(b \;\&\; P, v)\,;\, end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell}\\
&= \left(\begin{array}{l} get_p?l_p \to \cdots \to get_q?l_q \to (b \;\&\; F_{VarPost}\,(P, v))\\ ;end \to SKIP \end{array}\right) \mathbin{/\!\!/_{vs}} Rep_{MCell} && [F_{Var} \text{ Rule 8}]\\
&= \left(\begin{array}{l} \left(\begin{array}{l} b[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!]\\ \;\&\; F_{VarPost}\,(P[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!], v) \end{array}\right)\\ ;end \to SKIP \end{array}\right) \mathbin{/\!\!/_{vs}} Rep_{MCell} && [\text{Law C.1.3}]\\
&= \left(\begin{array}{l} b[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!] \;\&\\ ((F_{VarPost}\,(P[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!], v)\,;\, end \to SKIP) \mathbin{/\!\!/_{vs}} Rep_{MCell}) \end{array}\right) && [\text{Law C.1.13}]
\end{aligned}$$

According to assumption, the memory model of P terminates. Therefore, the boolean guard of it also terminates. In other words, the memory model of boolean guard $b \;\&\; P$ will not cause non-termination if the memory model of P terminate.

(10) If P is a parallel composition like $P_1 \underset{cs}{\parallel} P_2$, assume

$$\begin{aligned}
&scpV\,(P_1) \cap scpV\,(P_2) = \varnothing\\
&initRdLclV\,(P_1) = \{l_{p1}, \cdots, l_{q1}\}\\
&initRdLclV\,(P_2) = \{l_{p2}, \cdots, l_{q2}\}\\
&initRdLclV\,(P_1) \cup initRdLclV\,(P_2) = \{l_p, \cdots, l_q\}
\end{aligned}$$

and the memory models of both P_1 and P_2 terminate, then

$$\left(F_{Var}\left(P_1 \underset{cs}{\parallel} P_2, v\right); end \to SKIP\right) /\!\!/_{vs}\ Rep_{MCell}$$

$$= \left(\begin{array}{l} get_p?l_p \to \cdots \to get_q?l_q \to \left(F_{VarPost}\,(P_1, v) \underset{cs}{\parallel} F_{VarPost}\,(P_2, v)\right) \\ ;end \to SKIP \end{array}\right) /\!\!/_{vs}\ Rep_{MCell}$$

[F_{Var} Rule 9]

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}\,(P_1[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!]) \\ \underset{vs}{\parallel} F_{VarPost}\,(P_2[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!]) \end{array}\right) \\ ;end \to SKIP \end{array}\right) /\!\!/_{vs}\ Rep_{MCell}$$

[Law C.1.3]

$$= \left(\begin{array}{l} (F_{VarPost}\,(P_1[\![x_{p1}, \cdots, x_{q1}/l_{p1}, \cdots, l_{q1}]\!])\,; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell_{1..p}} \\ \underset{cs}{\parallel} \\ (F_{VarPost}\,(P_2[\![x_{p2}, \cdots, x_{q2}/l_{p2}, \cdots, l_{q2}]\!])\,; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell_{(p+1)..m}} \end{array}\right)$$

[Law C.1.10]

where $Rep_{MCell_{1..p}}$ and $Rep_{MCell_{(p+1)..m}}$ are partitioned memory processes.

According to assumption, the memory models of both P_1 and P_2 terminate. Therefore, the parallel composition of them also terminates. In other words, the memory model of parallel composition $P_1 \underset{cs}{\parallel} P_2$ will not cause non-termination if the memory models of P_1 and P_2 terminate.

(11) If P is an interleaving like $P_1 \mathbin{|||} P_2$, the proof is omitted since it is very similar to the proof for parallel composition in 10).

(12) If P is a recursion, it is not supported.

(13) If P is a replicated process, since replicated operators are just the expansion of corresponding operators, their proofs are very similar to corresponding operators and consequently omitted.

Finally, all F_{Var} rules except recursion will not result in non-termination if P terminates. Therefore,

$$(F_{Var}\,(P, v)\,; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell}$$

[Lemma C.1.1]

terminates as well according to Law C.1.20, and the lemma is proved. □

Basic CSP Actions

Lemma C.1.3 (Basic CSP Actions)**.** *The memory model of linked basic actions in CSP preserves the state and behaviour of basic actions in Circus.*

Proof. (1). The memory model of linked **Skip** is

$$\left(\begin{array}{l} (Us\,; F_{Var}\,(\Phi\,(R_{wrt}\,(\mathbf{Skip})), l)\,; end \to SKIP) \\ /\!\!/_{vs}\ Rep_{Mem} \end{array}\right) \quad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} (F_{Var}\,(\Phi\,(R_{wrt}\,(\mathbf{Skip})), l)\,; end \to SKIP) \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right) \quad \text{[Lemma C.1.1]}$$

$$= \left(\ (F_{Var}\,(SKIP, l)\,; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell}\ \right) \quad \text{[Link Rule 32]}$$

$$= \left(\ (SKIP\,; end \to SKIP) /\!\!/_{vs}\ Rep_{MCell}\ \right) \quad [F_{Var} \text{ Rule 2}]$$

$$= \left(\ (end \to SKIP) /\!\!/_{vs}\ Rep_{MCell}\ \right) \quad \text{[Roscoe [71, } \langle;\text{-unit-l}\rangle(7.2)]]$$

$$= \left(SKIP \underset{vs}{\|} \left(\underset{\{\!|end|\!\}}{\|} i : 1..m \bullet (SKIP) \right) \right) \setminus vs$$

[Roscoe [71, $\langle \|$ -step$\rangle$(3.10)] and hiding]

$$= (SKIP) \setminus vs \qquad \text{[Roscoe [71, Distributed Termination]]}$$

$$= SKIP$$

Therefore, the memory model of the linked **Skip** is $SKIP$. According to Section 5.10.2.1, they have the same semantics.

(2). The memory model of linked **Stop** is

$$\begin{aligned}
&\left(\begin{array}{l} (Us \,;\, F_{Var}\,(\Phi\,(R_{wrt}\,(\mathbf{Stop}))\,, l)\,;\, end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{Mem} \end{array} \right) && \text{[Definition B.1.5]}\\
&= \left(\ (F_{Var}\,(STOP, l)\,;\, end \to SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell} \ \right) && \text{[Link Rule 32]}\\
&= \left(\ (STOP\,;\, end \to SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell} \ \right) && [F_{Var} \text{ Rule 2]}\\
&= \left(\ (STOP) \mathbin{/\!\!/}_{vs} Rep_{MCell} \ \right) && \text{[Hoare [7, Section 5.2, L2]]}\\
&= STOP && \text{[Law C.1.1]}
\end{aligned}$$

Therefore, the memory model of the linked **Stop** is $STOP$. According to Section 5.10.2.1, they have the same semantics.

(3). The memory model of linked **Chaos** is

$$\begin{aligned}
&\left(\begin{array}{l} (Us \,;\, F_{Var}\,(\Phi\,(R_{wrt}\,(\mathbf{Chaos}))\,, l)\,;\, end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{Mem} \end{array} \right) && \text{[Definition B.1.5]}\\
&= \left(\ (F_{Var}\,(\mathbf{div}, l)\,;\, end \to SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell} \ \right) && \text{[Link Rule 32]}\\
&= \left(\ (\mathbf{div}\,;\, end \to SKIP) \mathbin{/\!\!/}_{vs} Rep_{MCell} \ \right) && [F_{Var} \text{ Rule 2]}\\
&= \left(\ (\mathbf{div}) \mathbin{/\!\!/}_{vs} Rep_{MCell} \ \right) && \text{[Roscoe [71, } \langle ;\text{-zero-l} \rangle (13.7)]]\\
&= \mathbf{div} && \text{[Law C.1.2]}
\end{aligned}$$

Therefore, the memory model of the linked **Chaos** is **div**. According to Section 5.10.2.1, they have the same semantics. □

Prefixing

Lemma C.1.4 (Prefixing)**.** *The memory model of linked prefixing in CSP preserves the state and behaviour of prefixing in Circus.*

Proof. (1). Provided a prefixing action

$$A \mathrel{\widehat{=}} c!e\,(l_p, \cdots, l_q)?l_k? \cdots ?l_n \to A'$$

where the message expression e evaluates local variables $l_p, \cdots, l_q$, and local variables $l_k, \cdots, l_n$ are input variables of c, then

$$\left(\begin{array}{l} (Us \,;\, F_{Var}\,(\Phi\,(R_{wrt}\,(A))\,, l)\,;\, end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{Mem} \end{array} \right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} (F_{Var}\,(\Phi\,(R_{wrt}\,(c!e\,(l_p, \cdots, l_q)?l_k? \cdots ?l_n \to A'))\,, l)\,;\, end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right)$$

[Lemma C.1.1]

$$= \left(\begin{array}{l} (F_{Var}\,(c!\Phi\,(e\,(l_p, \cdots, l_q))?l_k? \cdots ?l_n \to \Phi\,(R_{wrt}\,(A'))\,, l)\,;\, end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right)$$

[Link Rule 33 to 38]

$$= \left(\begin{array}{l}\left(\begin{array}{l} get_p?l_p \rightarrow \cdots \rightarrow get_q?l_q \rightarrow c!\Phi\left(e\left(l_p, \cdots, l_q\right)\right)?l_k?\cdots?l_n \rightarrow \\ set_k!l_k \rightarrow \cdots \rightarrow set_n!l_n \rightarrow F_{Var}\left(\Phi\left(R_{wrt}\left(A'\right)\right), l\right)\end{array}\right) \\ \quad ;end \rightarrow SKIP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(\begin{array}{ll} & set_i?y \rightarrow MCell_i\left(y\right) \\ \Box & get_i!x_i \rightarrow MCell_i\left(x_i\right) \\ \Box & end \rightarrow SKIP\end{array}\right)\right)\end{array}\right) \setminus vs$$

[F_{Var} Rule 3]

$$= \left(\begin{array}{l}\left(\begin{array}{l} c!\Phi\left(e\left(x_p, \cdots, x_q\right)\right)?l_k?\cdots?l_n \rightarrow \\ set_k!l_k \rightarrow \cdots \rightarrow set_n!l_n \rightarrow F_{Var}\left(\Phi\left(R_{wrt}\left(A'\right)\right), l\right)\end{array}\right) \\ \quad ;end \rightarrow SKIP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(\begin{array}{ll} & set_i?y \rightarrow MCell_i\left(y\right) \\ \Box & get_i!x_i \rightarrow MCell_i\left(x_i\right) \\ \Box & end \rightarrow SKIP\end{array}\right)\right)\end{array}\right) \setminus vs \qquad \text{[Law C.1.3]}$$

$$= \left(\begin{array}{l} c!\Phi\left(e\left(x_p, \cdots, x_q\right)\right)?l_k?\cdots?l_n \rightarrow \\ \left(\begin{array}{l} \left(set_k!l_k \rightarrow \cdots \rightarrow set_n!l_n \rightarrow F_{Var}\left(\Phi\left(R_{wrt}\left(A'\right)\right), l\right)\right) \\ \quad ;end \rightarrow SKIP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(\begin{array}{ll} & set_i?y \rightarrow MCell_i\left(y\right) \\ \Box & get_i!x_i \rightarrow MCell_i\left(x_i\right) \\ \Box & end \rightarrow SKIP\end{array}\right)\right)\end{array}\right)\end{array}\right) \setminus vs$$

[Roscoe [71, ⟨∥ -step⟩(3.10)]]

$$= \left(\begin{array}{l} c!\Phi\left(e\left(x_p, \cdots, x_q\right)\right)?l_k?\cdots?l_n \rightarrow \\ \left(\begin{array}{l} \left(F_{Var}\left(\Phi\left(R_{wrt}\left(A'\right)\right), l\right)\right); end \rightarrow SKIP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(\begin{cases} MCell_i\left(x_i\right) & i \notin k..n \\ MCell_i\left(l_i\right) & i \in k..n\end{cases}\right)\right)\end{array}\right)\end{array}\right) \setminus vs \qquad \text{[Law C.1.4]}$$

$$= \left(\begin{array}{l} c!\Phi\left(e\left(x_p, \cdots, x_q\right)\right)?l_k?\cdots?l_n \rightarrow \\ \left(\begin{array}{l} \left(F_{Var}\left(\Phi\left(R_{wrt}\left(A'\right)\right), l\right)\right); end \rightarrow SKIP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel} i : 1..m \bullet \left(\begin{cases} MCell_i\left(x_i\right) & i \notin k..n \\ MCell_i\left(l_i\right) & i \in k..n\end{cases}\right)\right)\end{array}\right) \setminus vs\end{array}\right) \qquad \text{[Hiding]}$$

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are prefixing.
- from the state aspect,
  - the local variables evaluated in the expression e of the communication c are substituted by their corresponding values in the memory. According to our assumptions, these values are equal to those in *Circus*. In addition, $\Phi\left(e\right)$ is a linked counterpart in CSP. Therefore, the $\Phi\left(e\left(x_p, \cdots, x_q\right)\right)$ is equal to e in *Circus*. Finally, the communication c in the transformed process above is equal to the communication c in *Circus*.

  - then for A', it is eventually transformed to the memory model of a linked process $\Phi\left(R_{wrt}\left(A'\right)\right)$ and the local variables that correspond to the input variables of c are updated to their input values in the memory, which is the same as that in *Circus* (the values of the input variables of c are seen by A').

- Eventually, the memory model of the linked process from a prefixing preserves the semantics of the prefixing in *Circus*.

$\square$

Guarded Action

Lemma C.1.5 (Guarded Action)**.** *The memory model of linked guarded action in CSP preserves the state and behaviour of the guarded action in Circus.*

Proof. Provided g evaluates local variables $l_{gp}, \cdots, l_{gq}$, and the first construct of A evaluates local variables $l_{ap}, \cdots, l_{aq}$. Furthermore, they together form a set of local variables $l_p, \cdots, l_q$.

$$\left(\begin{array}{l} \left(Us\,;F_{Var}\left(\Phi\left(R_{wrt}\left((g)\,\&\,A\right)\right),l\right);end \rightarrow SKIP\right) \\ \parallel_{vs}\ Rep_{Mem} \end{array}\right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} \left(\left(F_{Var}\left(\Phi\left(R_{wrt}\left((g)\,\&\,A\right)\right),l\right)\right);end \rightarrow SKIP\right) \\ \parallel_{vs}\ Rep_{MCell} \end{array}\right) \qquad \text{[Lemma C.1.1]}$$

$$= \left(\begin{array}{l} \left(F_{Var}\left(\left(\ \Phi(g)\ \ \&\ \ \Phi\left(R_{post}\left(A\right)\right)\ \right),l\right);end \rightarrow SKIP\right) \\ \parallel_{vs}\ Rep_{MCell} \end{array}\right) \qquad \text{[Link Rule 39]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{mrg}\left(F_{VarPre}\left(\Phi\left(g\right),l\right),F_{VarPre}\left(\Phi\left(R_{post}\left(A\right)\right),l\right)\right) \rightarrow \\ \left(\Phi\left(g\right)\ \ \&\ \ F_{VarPost}\left(\Phi\left(R_{post}\left(A\right)\right)\right),l\right);end \rightarrow SKIP \end{array}\right) \\ \parallel_{vs}\ Rep_{MCell} \end{array}\right)$$

$$\text{[}F_{Var}\text{ Rule 3 and }F_{Var}\text{ Rule 8]}$$

$$= \left(\left(\begin{array}{l} \left(\begin{array}{l} get_p?l_p \rightarrow \cdots \rightarrow get_q?l_q \rightarrow \\ \left(\Phi\left(g\right)\ \ \&\ \ F_{VarPost}\left(\Phi\left(R_{post}\left(A\right)\right)\right),l\right);end \rightarrow SKIP \end{array}\right) \\ \parallel_{vs}\ Rep_{MCell} \end{array}\right)\right)$$

$$\text{[}g\text{ and the first construct of }A\text{ evaluate }l_p, \cdots, l_q\text{]}$$

$$= \left(\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \Phi\left(g[x_{gp}, \cdots, x_{gq}/l_{gp}, \cdots, l_{gq}]\right)\ \ \& \\ F_{VarPost}\left(\Phi\left(R_{post}\left(A\llbracket x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}\rrbracket\right)\right)\right),l \end{array}\right) \\ ;end \rightarrow SKIP \end{array}\right) \\ \parallel_{vs}\ Rep_{MCell} \end{array}\right)\right)$$

$$\text{[Law C.1.1.4]}$$

$$= \left(\begin{array}{l} \Phi\left(g[x_{gp}, \cdots, x_{gq}/l_{gp}, \cdots, l_{gq}]\right)\ \ \& \\ \left(\begin{array}{l} \left(\begin{array}{l} \left(\ F_{VarPost}\left(\Phi\left(R_{post}\left(A\llbracket x_{ap}, \cdots, x_{aq}/l_{ap}, \cdots, l_{aq}\rrbracket\right)\right)\right),l\ \right) \\ ;end \rightarrow SKIP \end{array}\right) \\ \parallel_{vs}\ Rep_{MCell} \end{array}\right) \end{array}\right)$$

[g does not evaluate local variables since they have been substituted, and Law C.1.11]

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are boolean guards.

- from the state aspect,

  - the local variables evaluated in the condition g are substituted by their corresponding values in the memory. According to our assumptions, these values are equal to those in *Circus*. In addition, $\Phi(g)$ is a linked counterpart in CSP. Therefore, the $\Phi(g[x_{gp}, \cdots, x_{gq}/l_{gp}, \cdots, l_{gq}])$ is equal to g in *Circus*. Finally, the condition g in the transformed process above is equal to the g in *Circus*.

– then for A, it is eventually transformed to the memory model of a linked process and the local variables in the memory are left unchanged, which is the same as that in *Circus* (the action A evaluates the same values of local variables as the condition g).

- Eventually, the memory model of the linked process from a guarded action preserves the semantics of that in *Circus*.

□

Sequential Composition

Lemma C.1.6 (Sequential Composition). *The memory model of linked sequential composition preserves the state and behaviour.*

Proof. Provided the local variables $l_p, \cdots, l_q$ are evaluated in the first construct of the first action A_1,

$$\left(\begin{array}{l} (Us \mathbin{;} F_{Var}(\Phi(R_{wrt}(A_1 \mathbin{;} A_2)), l) \mathbin{;} end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{Mem} \end{array}\right) \qquad [\text{Definition B.1.5}]$$

$$= \left(\begin{array}{l} (F_{Var}(\Phi(R_{wrt}(A_1 \mathbin{;} A_2)), l) \mathbin{;} end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array}\right) \qquad [\text{Lemma C.1.1}]$$

$$= \left(\begin{array}{l} (F_{Var}((\Phi(R_{post}(A_1)) \mathbin{;} \Phi(R_{wrt}(A_2))), l) \mathbin{;} end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array}\right) \qquad [\text{Link Rule 40}]$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPre}(\Phi(R_{post}(A_1)), l) \to \\ (F_{VarPost}(\Phi(R_{post}(A_1)), l) \mathbin{;} F_{Var}(\Phi(R_{wrt}(A_2)), l)) \\ \mathbin{;} end \to SKIP \end{array}\right) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array}\right) \qquad [F_{Var} \text{ Rule 4}]$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} get_p?l_p \to \cdots \to get_q?l_q \to \\ (F_{VarPost}(\Phi(R_{post}(A_1)), l) \mathbin{;} F_{Var}(\Phi(R_{wrt}(A_2)), l)) \\ \mathbin{;} end \to SKIP \end{array}\right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel}\ i : 1..m \bullet MCell_i(x_i)\right) \end{array}\right) \setminus vs$$

[Definition B.1.3]

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}(\Phi(R_{post}(A_1[\![x_p, \cdots, x_q/l_p, \cdots, l_q]\!])), l) \\ \mathbin{;} F_{Var}(\Phi(R_{wrt}(A_2)), l) \end{array}\right) \\ \mathbin{;} end \to SKIP \end{array}\right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel}\ i : 1..m \bullet MCell_i(x_i)\right) \end{array}\right) \setminus vs$$

[Law C.1.1.4]

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost} \left(\Phi \left(R_{post} \left(A_1 [\![x_p, \cdots, x_q / l_p, \cdots, l_q]\!] \right) \right), l \right) \\ ; \left(F_{Var} \left(\Phi \left(R_{wrt} \left(A_2 \right) \right), l \right) ; end \to SKIP \right) \end{array} \right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!| end |\!\}}{\parallel} i : 1..m \bullet MCell_i \left(x_i \right) \right) \end{array} \right) \setminus vs$$

[Roscoe [71, ⟨;-assoc⟩(7.3)]]

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost} \left(\Phi \left(R_{wrt} \left(A_1 [\![x_p, \cdots, x_q / l_p, \cdots, l_q]\!] \right) \right), l \right) \\ ; end \to SKIP \end{array} \right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!| end |\!\}}{\parallel} i : 1..m \bullet \left(MCell_i \left(x_i \right) \right) \right) \end{array} \right) \setminus vs \\ ; \\ \left(\begin{array}{l} \left(F_{Var} \left(\Phi \left(R_{wrt} \left(A_2 \right) \right), l \right) ; end \to SKIP \right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!| end |\!\}}{\parallel} i : 1..m \bullet \left(\begin{cases} MCell_i \left(x_i \right) & i \notin j..k \\ MCell_i \left(l_i \right) & i \in j..k \end{cases} \right) \right) \end{array} \right) \setminus vs \end{array} \right)$$

[Law C.1.14]

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are sequential composition.
- from the state aspect,
  - the local variables evaluated in the first construct of the first action A_1 are substituted by their corresponding values in the memory. According to our assumptions, these values are equal to those in *Circus*. In addition, the first process in the transformed sequential composition is a memory model of the linked process from A_1 in *Circus*, and its memory is initially equal to the local variables in *Circus*.
  - then the second process in the transformed sequential composition is a memory model of the linked process from A_2 in *Circus*, and its memory is another copy of the initial memory but all updates by the first process are seen. That is the same as the sequential composition in *Circus*.
- Eventually, the memory model of the linked process from a sequential composition preserves the semantics of that in *Circus*.

□

External Choice

Lemma C.1.7 (External Choice)**.** *The memory model of linked external choice of actions preserves the state and behaviour.*

Proof. Provided the local variables $l_{p_1}, \cdots, l_{q_1}$ are evaluated in the first construct of the first action AA_1, and the local variables $l_{p_2}, \cdots, l_{q_2}$ are evaluated in the first construct of the second action AA_2.

$$initRdLclV \left(AA_1 \right) = \{ l_{p_1}, \cdots, l_{q_1} \}$$
$$initRdLclV \left(AA_2 \right) = \{ l_{p_2}, \cdots, l_{q_2} \}$$

Then

$$\left(\begin{array}{l} (Us\,; F_{Var}\left(\Phi\left(R_{wrt}\left(AA_1 \Box AA_2\right)\right), l\right); end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{Mem} \end{array} \right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} (F_{Var}\left(\Phi\left(R_{wrt}\left(AA_1 \Box AA_2\right)\right), l\right); end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right) \qquad \text{[Lemma C.1.1]}$$

$$= \left(\begin{array}{l} (F_{Var}\left(\Phi\left(R_{post}\left(AA_1\right)\right) \Box \Phi\left(R_{post}\left(AA_2\right)\right), l\right); end \to SKIP) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right)$$

[Link Rule 41]

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{mrg}\left(\begin{array}{l} F_{VarPre}\left(\Phi\left(R_{post}\left(AA_1\left(l_{p1}, \cdots, l_{q1}\right)\right)\right), l\right) \\ F_{VarPre}\left(\Phi\left(R_{post}\left(AA_2\left(l_{p2}, \cdots, l_{q2}\right)\right)\right), l\right) \end{array} \right) \\ \to \left(\begin{array}{l} F_{VarPost}\left(\Phi\left(R_{post}\left(AA_1\left(l_{p1}, \cdots, l_{q1}\right)\right)\right), l\right) \\ \Box \\ F_{VarPost}\left(\Phi\left(R_{post}\left(AA_2\left(l_{p2}, \cdots, l_{q2}\right)\right)\right), l\right) \end{array} \right) \\ ;end \to SKIP \end{array} \right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel}\ i : 1..m \bullet MCell_i\left(x_i\right) \right) \end{array} \right) \setminus vs$$

[F_{Var} Rule 6]

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}\left(\Phi\left(R_{post}\left(AA_1[\![x_{p1}, \cdots, x_{q1} / l_{p1}, \cdots, l_{q1}]\!]\right)\right), l\right) \\ \Box \\ F_{VarPost}\left(\Phi\left(R_{post}\left(AA_2[\![x_{p2}, \cdots, x_{q2} / l_{p2}, \cdots, l_{q2}]\!]\right)\right), l\right) \end{array} \right) \\ ;end \to \mathbf{Skip} \end{array} \right) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right)$$

[Defintion B.1.3, Definition B.1.4, and Law C.1.3]

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}\left(\Phi\left(R_{post}\left(AA_1[\![x_{p1}, \cdots, x_{q1} / l_{p1}, \cdots, l_{q1}]\!]\right)\right), l\right) \\ ;end \to SKIP \end{array} \right) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right) \\ \Box \\ \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}\left(\Phi\left(R_{post}\left(AA_2[\![x_{p2}, \cdots, x_{q2} / l_{p2}, \cdots, l_{q2}]\!]\right)\right), l\right) \\ ;end \to SKIP \end{array} \right) \\ \mathbin{/\!\!/}_{vs}\ Rep_{MCell} \end{array} \right) \end{array} \right)$$

[Law C.1.8]

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are external choice.
- from the state aspect,
  - both the memory model of the linked process from AA_1 and the memory model of the linked process from AA_2 in the transformed external choice are based on the same memory. That is the same as external choice of actions in *Circus* where AA_1 and AA_2 see a same copy of local variables. And these local variables are equal to local variables in the memory of CSP.
- Eventually, the memory model of the linked process from an external choice preserves the semantics of that in *Circus*.

□

Internal Choice

Lemma C.1.8 (Internal Choice)**.** *The memory model of linked internal choice of actions preserves the state and behaviour.*

Proof. Provided the local variables $l_{p_1}, \cdots, l_{q_1}$ are evaluated in the first construct of the first action A_1, and the local variables $l_{p_2}, \cdots, l_{q_2}$ are evaluated in the first construct of the second action A_2. Then

$$\left(\begin{array}{l} (Us \; ; F_{Var}(\Phi(R_{wrt}(A_1 \sqcap A_2)), l) \; ; end \rightarrow SKIP) \\ \mathbin{/\!\!/}_{vs} \; Rep_{Mem} \end{array}\right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} ((F_{Var}(\Phi(R_{wrt}(A_1 \sqcap A_2)), l)) \; ; end \rightarrow SKIP) \\ \mathbin{/\!\!/}_{vs} \; Rep_{MCell} \end{array}\right) \qquad \text{[Lemma C.1.1]}$$

$$= \left(\begin{array}{l} (F_{Var}(\Phi(R_{post}(A_1)) \sqcap \Phi(R_{post}(A_2)), l) \; ; end \rightarrow SKIP) \\ \mathbin{/\!\!/}_{vs} \; Rep_{MCell} \end{array}\right) \qquad \text{[Link Rule 42]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{mrg}\left(\begin{array}{l} F_{VarPre}(\Phi(R_{post}(A_1(l_{p_1}, \cdots, l_{q_1}))), l) \\ F_{VarPre}(\Phi(R_{post}(A_2(l_{p_2}, \cdots, l_{q_2}))), l) \end{array}\right) \\ \rightarrow \left(\begin{array}{l} F_{VarPost}(\Phi(R_{post}(A_1(l_{p_1}, \cdots, l_{q_1}))), l) \\ \sqcap \\ F_{VarPost}(\Phi(R_{post}(A_2(l_{p_2}, \cdots, l_{q_2}))), l) \end{array}\right) \\ ;end \rightarrow SKIP \end{array}\right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel} \; i : 1..m \bullet MCell_i(x_i)\right) \end{array}\right) \setminus vs$$

$$\text{[}F_{Var}\text{ Rule 7]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}(\Phi(R_{post}(A_1[\![x_{p_1}, \cdots, x_{q_1}/l_{p_1}, \cdots, l_{q_1}]\!])), l) \\ \sqcap \\ F_{VarPost}(\Phi(R_{post}(A_2[\![x_{p_2}, \cdots, x_{q_2}/l_{p_2}, \cdots, l_{q_2}]\!])), l) \end{array}\right) \\ ;end \rightarrow \mathbf{Skip} \end{array}\right) \\ \mathbin{/\!\!/}_{vs} \; Rep_{MCell} \end{array}\right)$$

$$\text{[Defintion B.1.3, Definition B.1.4, and Law C.1.3]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}(\Phi(R_{post}(A_1[\![x_{p_1}, \cdots, x_{q_1}/l_{p_1}, \cdots, l_{q_1}]\!])), l) \\ ;end \rightarrow SKIP \end{array}\right) \\ \mathbin{/\!\!/}_{vs} \; Rep_{MCell} \end{array}\right) \\ \sqcap \\ \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}(\Phi(R_{post}(A_2[\![x_{p_2}, \cdots, x_{q_2}/l_{p_2}, \cdots, l_{q_2}]\!])), l) \\ ;end \rightarrow SKIP \end{array}\right) \\ \mathbin{/\!\!/}_{vs} \; Rep_{MCell} \end{array}\right) \end{array}\right)$$

$$\text{[Law C.1.10]}$$

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are internal choice.
- from the state aspect,
  - both the memory model of the linked process from A_1 and the memory model of the linked process from A_2 in the transformed internal choice are based on the same memory. That is the same as internal choice of actions in *Circus* where A_1 and A_2 see a same copy of local variables. And these local variables are equal to local variables in the memory of CSP.

- Eventually, the memory model of the linked process from an internal choice preserves the semantics of that in *Circus*.

$\Box$

Schema Expression

Lemma C.1.9 (Schema Expression)**.** *The memory model of linked schema expression as action preserves the state and behaviour.*

Proof.

$$\left(\begin{array}{l} (Us\ ;\ F_{Var}\left(\Phi\left(R_{wrt}\left(SExp\right)\right), l\right); end \to SKIP) \\ /\!\!/_{vs}\ Rep_{Mem} \end{array}\right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} (F_{Var}\left(\Phi\left(R_{wrt}\left(SExp\right)\right), l\right); end \to SKIP) \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right) \qquad \text{[Lemma C.1.1]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{Var}\left(\left(\begin{array}{l} P\_SExp!l_p!\cdots!l_q?l_k?\cdots?l_n \to SKIP \\ \Box \\ P\_SExp\_fOp!l_p!\cdots!l_q \to \mathbf{div} \end{array}\right), l\right) \\ ;end \to SKIP \end{array}\right) \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right)$$

[Link Rule 31]

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{mrg}\left(\begin{array}{l} F_{VarPre}\left(P\_SExp!l_p!\cdots!l_q?l_k?\cdots?l_n \to SKIP, l\right), \\ F_{VarPre}\left(P\_SExp\_fOp!l_p!\cdots!l_q \to \mathbf{div}, l\right) \end{array}\right) \\ \to \left(\begin{array}{l} F_{VarPost}\left(P\_SExp!l_p!\cdots!l_q?l_k?\cdots?l_n \to SKIP, l\right) \\ \Box \\ F_{VarPost}\left(P\_fSExp!l_p!\cdots!l_q \to \mathbf{div}, l\right) \end{array}\right) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel}\ i : 1..m \bullet \left(MCell_i\left(x_i\right)\right)\right) \end{array}\right) \setminus vs$$

[F_{Var} Rule 6]

$$= \left(\begin{array}{l} \left(\begin{array}{l} get_p?l_p \to \cdots \to get_q?l_q \to \\ \quad \left(\begin{array}{l} P\_SExp!l_p!\cdots!l_q?l_k?\cdots?l_n \to \\ \qquad set_k!l_k \to \cdots \to set_n!l_n \to SKIP \\ \Box \\ P\_fSExp!l_p!\cdots!l_q \to \mathbf{div} \end{array}\right) \\ ;end \to SKIP \end{array}\right) \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right)$$

[Definition B.1.4, and F_{Var} Rule 3]

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} P\_SExp!x_p!\cdots!x_q?l_k?\cdots?l_n \to \\ \qquad set_k!l_k \to \cdots \to set_n!l_n \to SKIP \\ \Box \\ P\_SExp\_fOp!x_p!\cdots!x_q \to \mathbf{div} \end{array}\right) \\ ;end \to SKIP \end{array}\right) \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right) \qquad \text{[Law C.1.3]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} P\_SExp!x_p!\cdots!x_q?l_k?\cdots?l_n \rightarrow \\ \left(\begin{array}{l} set_k!l_k \rightarrow \cdots \rightarrow set_n!l_n \rightarrow SKIP\,;\, end \rightarrow SKIP \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right) \end{array}\right) \\ \Box \\ \left(\begin{array}{l} P\_fSExp!x_p!\cdots!x_q \rightarrow \\ \left(\begin{array}{l} \mathbf{div}\,;\, end \rightarrow SKIP \\ /\!\!/_{vs}\ Rep_{MCell} \end{array}\right) \end{array}\right) \end{array}\right)$$

[Law C.1.8]

$$= \left(\begin{array}{l} \left(\begin{array}{l} P\_SExp!x_p!\cdots!x_q?l_k?\cdots?l_n \rightarrow \\ \left(\begin{array}{l} end \rightarrow SKIP \\ \underset{vs}{\parallel} \\ \left(\underset{\{\!|end|\!\}}{\parallel}\ i : 1..m \bullet \left(\left\{\begin{array}{ll} MCell_i\,(x_i) & i \notin k..n \\ MCell_i\,(l_i) & i \in k..n \end{array}\right)\right)\right) \end{array}\right) \setminus vs \end{array}\right) \\ \Box \\ \left(\ P\_SExp\_fOp!x_p!\cdots!x_q \rightarrow \mathbf{div}\ \right) \end{array}\right)$$

[Law C.1.4, [71, ⟨;-zero-l⟩(13.7)] and Law C.1.2]

$$= \left(\begin{array}{l} P\_SExp!x_p!\cdots!x_q?l_k?\cdots?l_n \rightarrow SKIP \\ \Box \\ P\_SExp\_fOp!x_p!\cdots!x_q \rightarrow \mathbf{div} \end{array}\right) \qquad \text{[Lemma C.1.2]}$$

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are external choice.
- from the state aspect,
  - the local variables evaluated in the first construct of both prefixing processes in the transformed external choice are substituted by their corresponding values in the memory. According to our assumptions, these values are equal to those in *Circus*.
- Eventually, the memory model of the linked process from a schema expression preserves the semantics of that in *Circus*.

□

Parallel Composition (Disjoint Variables in Scope)

Lemma C.1.10 (Parallel Composition (Disjoint Variables in Scope)). *The memory model of linked parallel composition of actions (in case of disjoint variables in scope) preserves the state and behaviour.*

Proof. Provided

$$scpV\,(A_1) \cap scpV\,(A_2) = \varnothing$$
$$lclV\,(A_1) = \{l_1, \ldots, l_p\}$$
$$lclV\,(A_2) = \{l_{p+1}, \ldots, l_m\}$$

and the local variables $l_{p_1}, \cdots, l_{q_1}$ are evaluated in the first construct of the first action A_1, and the local variables $l_{p_2}, \cdots, l_{q_2}$ are evaluated in the first construct of the second action

A_2. Then

$$\left(\begin{array}{l}\left(\begin{array}{l}Us\,;F_{Var}\left(\Phi\left(R_{wrt}\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right)\right), l\right) \\ ;end \to SKIP\end{array}\right) \\ \parallel_{vs}\ Rep_{Mem}\end{array}\right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l}\left(F_{Var}\left(\Phi\left(R_{wrt}\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right)\right), l\right); end \to SKIP\right) \\ \parallel_{vs}\ Rep_{MCell}\end{array}\right)$$

[Lemma C.1.1]

$$= \left(\begin{array}{l}\left(F_{Var}\left(\Phi\left(R_{post}\left(A_1\right)\right) \underset{\Phi(cs)}{\parallel} \Phi\left(R_{post}\left(A_2\right)\right), l\right); end \to SKIP\right) \\ \parallel_{vs}\ Rep_{MCell}\end{array}\right)$$

[Link Rule 43]

$$= \left(\begin{array}{l}\left(\begin{array}{l}F_{mrg}\left(\begin{array}{l}F_{VarPre}\left(\Phi\left(R_{post}\left(A_1\left(l_{p_1},\cdots,l_{q_1}\right)\right)\right), l\right) \\ F_{VarPre}\left(\Phi\left(R_{post}\left(A_2\left(l_{p_2},\cdots,l_{q_2}\right)\right)\right), l\right)\end{array}\right) \\ \to \left(\begin{array}{l}F_{VarPost}\left(\Phi\left(R_{post}\left(A_1\left(l_{p_1},\cdots,l_{q_1}\right)\right)\right), l\right) \\ \underset{\Phi(cs)}{\parallel} \\ F_{VarPost}\left(\Phi\left(R_{post}\left(A_2\left(l_{p_2},\cdots,l_{q_2}\right)\right)\right), l\right)\end{array}\right) \\ ;end \to SKIP\end{array}\right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel}\ i : 1..m \bullet MCell_i\left(x_i\right)\right)\end{array}\right) \setminus vs$$

[F_{Var} Rule 9]

$$= \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}F_{VarPost}\left(\Phi\left(R_{post}\left(A_1\llbracket x_{p_1},\cdots,x_{q_1}/l_{p_1},\cdots,l_{q_1}\rrbracket\right)\right), l\right) \\ \underset{\Phi(cs)}{\parallel} \\ F_{VarPost}\left(\Phi\left(R_{post}\left(A_2\llbracket x_{p_2},\cdots,x_{q_2}/l_{p_2},\cdots,l_{q_2}\rrbracket\right)\right), l\right)\end{array}\right) \\ ;end \to \mathbf{Skip}\end{array}\right) \\ \parallel_{vs}\ Rep_{MCell}\end{array}\right)$$

[Defintion B.1.3, Definition B.1.4, and Law C.1.3]

$$= \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}F_{VarPost}\left(\Phi\left(R_{post}\left(A_1\llbracket x_{p_1},\cdots,x_{q_1}/l_{p_1},\cdots,l_{q_1}\rrbracket\right)\right), l\right) \\ ;end \to SKIP\end{array}\right) \\ \parallel_{vs}\ Rep_{MCell_1}\end{array}\right) \\ \underset{\Phi(cs)}{\parallel} \\ \left(\begin{array}{l}\left(\begin{array}{l}F_{VarPost}\left(\Phi\left(R_{post}\left(A_2\llbracket x_{p_2},\cdots,x_{q_2}/l_{p_2},\cdots,l_{q_2}\rrbracket\right)\right), l\right) \\ ;end \to SKIP\end{array}\right) \\ \parallel_{vs}\ Rep_{MCell_2}\end{array}\right)\end{array}\right)$$

[Law C.1.17]

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are parallel composition on $\phi\left(cs\right)$.
- from the state aspect,
  - the memory model of the linked process from A_1 is based on one partition of the memory (from l_1 to l_p) and the memory model of the linked process from

A_2 is based on another partition of the memory (from l_{p+1}, to l_m). That is the same as parallel composition of actions in *Circus* (in case of disjoint variables in scope) where A_1 and A_2 have disjoint local variables.

- Eventually, the memory model of the linked process from a parallel composition preserves the semantics of that in *Circus*.

□

Interleaving (Disjoint Variables in Scope)

Lemma C.1.11 (Inteleaving (Disjoint Variables in Scope)). *The memory model of linked interleaving of actions (in case of disjoint variables in scope) preserves the state and behaviour.*

Proof. The interleaving of actions is very similar to the parallel composition of actions and its proof is omitted.

□

Hiding

Lemma C.1.12 (Hiding). *The memory model of linked hiding of action preserves the state and behaviour.*

Proof. Provided the local variables $l_p, \cdots, l_q$ are evaluated in the first construct of the action A, then

$$\left(\begin{array}{l} \left(Us\,;F_{Var}\left(\Phi\left(R_{wrt}\left(A \setminus cs\right)\right), l\right); end \to SKIP\right) \\ \parallel_{vs}\; Rep_{Mem} \end{array}\right) \qquad \text{[Definition B.1.5]}$$

$$= \left(\begin{array}{l} \left(F_{Var}\left(\Phi\left(R_{wrt}\left(A \setminus cs\right)\right), l\right); end \to SKIP\right) \\ \parallel_{vs}\; Rep_{MCell} \end{array}\right) \qquad \text{[Lemma C.1.1]}$$

$$= \left(\begin{array}{l} \left(F_{Var}\left(\left(\Phi\left(R_{post}\left(A\right)\right) \setminus \Phi\left(cs\right)\right), l\right); end \to SKIP\right) \\ \parallel_{vs}\; Rep_{MCell} \end{array}\right) \qquad \text{[Link Rule 47]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} F_{VarPre}\left(\Phi\left(R_{post}\left(A\right)\right)\right) \to \\ \left(F_{VarPost}\left(\Phi\left(R_{post}\left(A\right)\right), l\right) \setminus \Phi\left(cs\right)\right) \\ ;end \to SKIP \end{array}\right) \\ \parallel_{vs}\; Rep_{MCell} \end{array}\right) \qquad \text{[}F_{Var}\text{ Rule 5]}$$

$$= \left(\begin{array}{l} \left(\begin{array}{l} get_p?l_p \to \cdots \to get_q?l_q \to \\ \left(F_{VarPost}\left(\Phi\left(R_{post}\left(A\right)\right), l\right) \setminus \Phi\left(cs\right)\right) \\ ;end \to SKIP \end{array}\right) \\ \underset{vs}{\parallel} \\ \left(\underset{\{\mid end \mid\}}{\parallel}\; i : 1..m \bullet MCell_i\left(x_i\right)\right) \end{array}\right) \setminus vs$$

[Definition B.1.4, and F_{Var} Rule 3]

$$= \left(\begin{array}{l} \left(\begin{array}{l} \left(F_{VarPost}\left(\Phi\left(R_{post}\left(A[\![x_p, \cdots, x_q / l_p, \cdots, l_q]\!]\right)\right), l\right) \setminus \Phi\left(cs\right)\right) \\ ;end \to SKIP \end{array}\right) \\ \parallel_{vs}\; Rep_{MCell} \end{array}\right)$$

[Law C.1.3]

$$= \left(\begin{array}{l} \left(\left(\begin{array}{l} F_{VarPost}\left(\Phi\left(R_{post}\left(A[\![x_p, \cdots, x_q / l_p, \cdots, l_q]\!]\right)\right), l\right) \\ ;end \to SKIP \end{array}\right) \setminus \Phi\left(cs\right)\right) \\ \parallel_{vs}\; Rep_{MCell} \end{array}\right)$$

[Law C.1.16]

$$= \left(\left(\begin{array}{l} \left(\begin{array}{l} F_{VarPost}\left(\Phi\left(R_{post}\left(A[\![x_p, \cdots, x_q / l_p, \cdots, l_q]\!]\right)\right), l\right) \\ ; end \rightarrow SKIP \end{array} \right) \\ /\!\!/_{vs}\ Rep_{MCell} \end{array} \right) \right) \setminus \Phi\left(cs\right)$$

[Law C.1.15]

Therefore,

- from the behavioural aspect, the transformed process above has the same syntax as the linked process, and both are hiding from $\phi(cs)$.
- from the state aspect,
  - the memory model of the linked process from A is based on the same memory as the local variables l in *Circus*,
- Eventually, the memory model of the linked process from a hiding preserves the semantics of that in *Circus*.

□

Recursion The memory model of linked recursion of an action has not been proved to preserve the state and behaviour. Finally, we exclude it from the memory model, or recursion is not supported within a variable block in *Circus*.

Iterated Sequential Composition

Lemma C.1.13 (Iterated Sequential Composition)**.** *The memory model of linked iterated sequential composition preserves the state and behaviour.*

Proof. Since the iterated sequential composition $\mathop{;}$ is just an expansion of the corresponding replicated operator ;, the proof of this lemma is very similar to Lemma C.1.6 and therefore omitted here. □

Iterated External Choice

Lemma C.1.14 (Iterated External Choice)**.** *The memory model of linked iterated external choice preserves the state and behaviour.*

Proof. Since the iterated sequential composition $\Box$ is just an expansion of the corresponding replicated operator $\Box$, the proof of this lemma is very similar to Lemma C.1.7 and therefore omitted here. □

Iterated Internal Choice

Lemma C.1.15 (Iterated Internal Choice)**.** *The memory model of linked iterated internal choice preserves the state and behaviour.*

Proof. Since the iterated sequential composition $\sqcap$ is just an expansion of the corresponding replicated link $\sqcap$, the proof of this lemma is very similar to Lemma C.1.8 and therefore omitted here. □

Assignment

Lemma C.1.16 (Assignment)**.** *The memory model of linked assignment preserves the state and behaviour.*

Proof. According to Link Rule 52, an assignment is rewritten to a schema expression as action which is linked to the resultant $CSP \parallel_B Z$. Since the memory model of the linked process from a schema expression as action preserves the state and behaviour by Lemma C.1.9, the memory model of the linked process from assignment also preserves the state and behaviour. □

Alternation

Lemma C.1.17 (Alternation). *The memory model of linked alternation preserves the state and behaviour.*

Proof. According to Link Rule 53, an alternation is rewritten to an external choice of guarded actions which is linked to the resultant $\mathit{CSP} \parallel_B Z$. Since the memory models of the linked processes from external choice, guarded action, and internal choice preserve the state and behaviour by Lemma C.1.7, Lemma C.1.5, and Lemma C.1.8, the memory model of the linked process from alternation also preserves the state and behaviour. □

Variable Block

Lemma C.1.18 (Variable Block). *The memory model of linked variable block preserves the state and behaviour.*

Proof.

$$
\begin{aligned}
& F_{Mem}\left(\Phi\left(R_{wrt}\left(\mathbf{var}\, y : T \bullet A\right)\right), l\right) \\
& = F_{Mem}\left(\sqcap_{y:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{Post}(A)\right), \{y\}\right), l\right) && \text{[Link Rule 54]} \\
& = \sqcap_{y:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{Post}(A)\right), \{y\} \cup l\right) && \text{[Law C.1.19]}
\end{aligned}
$$

Therefore, the memory model of the linked process from an embedded variable block is equal to the memory model of the same process but with combined memory (combination of current memory with the memory from the variable block). Therefore, the memory model of linked variable block preserves the state and behaviour. □

Specification Statement

Lemma C.1.19 (Specification Statement). *The memory model of linked specification statement preserves the state and behaviour.*

Proof. According to Link Rule 55, a specification $w : [\mathit{pre}, \mathit{post}]$ is rewritten to a schema expression as action which is linked to the $\mathit{CSP} \parallel_B Z$. In particular, the dashed variables in *post* but not in w are hidden by additional existential quantification. Because the semantics of the schema expression is given by the specification statement ([35, Definition B.40]) and the linked schema expression preserves the state and behaviour (Lemma C.1.9), the linked specification statement preserves the state and behaviour. □

Assumption

Lemma C.1.20 (Assumption). *The memory model of linked assumption preserves the state and behaviour.*

Proof. An assumption $\{\mathit{pre}\}$ is equal to a specification $w : [\mathit{pre}, \mathit{true}]$ in which w is empty and *post* is *true*. Therefore according to Lemma C.1.19, the memory model of linked assumption preserves the state and behaviour. □

Coercion

Lemma C.1.21 (Coercion). *The memory model of linked coercion preserves the state and behaviour.*

Proof. A coercion $[\mathit{post}]$ is equal to a specification $w : [\mathit{true}, \mathit{post}]$ in which w is empty, that means no variables changed, and *pre* is *true*. In particular, the dashed variables in *post* are hidden by additional existential quantification. Therefore according to Lemma C.1.19, the memory model of linked coercion preserves the state and behaviour. □

Parametrisation By Value

Lemma C.1.22 (Parametrisation By Value)**.** *The memory model of linked parametrisation by value preserves the state and behaviour.*

Proof. A parametrisation by value, $((\mathbf{val}\, x : T \bullet A)\,(e))$, is written to a variable block with x initialized to e before the action A indeed according to Link Rule 58. Since variable block, sequential composition and assignment preserve the state and behaviour by Lemma C.1.18, C.1.6, and C.1.16, the memory model of linked parametrisation by value preserves the state and behaviour. □

Parametrisation By Result

Lemma C.1.23 (Parametrisation By Result)**.** *The memory model of linked parametrisation by result preserves the state and behaviour.*

Proof. A parametrisation by result, $((\mathbf{res}\, x : T \bullet A)\,(y))$, is written to a variable block with y set to the value of x after the action A indeed according to Link Rule 59. Since variable block, sequential composition and assignment preserve the state and behaviour by Lemma C.1.18, C.1.6, and C.1.16, the memory model of linked parametrisation by result preserves the state and behaviour. □

Parametrisation By Value-Result

Lemma C.1.24 (Parametrisation By Value-Result)**.** *The memory model of linked parametrisation by value-result preserves the state and behaviour.*

Proof. A parametrisation by value-result, $((\mathbf{vres}\, x : T \bullet A)\,(y))$, is written to a variable block with x initialized to the value of y before the action A and y set to the value of x after the action A indeed according to Link Rule 60. Since variable block, sequential composition and assignment preserve the state and behaviour by Lemma C.1.18, C.1.6 and C.1.16, the memory model of linked parametrisation by value-result preserves the state and behaviour. □

Renaming

Lemma C.1.25 (Renaming)**.** *The memory model of linked renaming preserves the state and behaviour.*

Proof. The action renaming is just a syntactical replacement of the name of variables. □

Action Invocation

Lemma C.1.26 (Action Invocation)**.** *The memory model of linked action invocation preserves the state and behaviour.*

Proof. The semantics of an action invocation is given by the copy rule and is simply equal to its body according to Link Rule 62. So the memory model of linked action invocation is the same as the memory model of its linked action body. Therefore, the memory model of linked action invocation preserves the state and behaviour. □

Parametrised Action

Lemma C.1.27 (Parametrised Action)**.** *The memory model of linked parametrised action invocation preserves the state and behaviour.*

Proof. The parametrised action invocation is defined by the copy rule and the substitution rule, and it is simply equal to its action body with parameters substituted by expressions according to Link Rule 65. So the memory model of the linked parametrised action invocation is the same as the memory model of its linked and substituted action body. Therefore, the memory model of linked parametrised action invocation preserves the state and behaviour. □

C.1.3 Theorems

C.1.3.1 Memory Model

Theorem C.1.28. The memory model of the CSP process $\Phi\left(R_{wrt}(A)\right)$ which is linked from the action A,

$$F_{Mem}\left(\Phi\left(R_{wrt}(A)\right), l\right)$$

where A is a *Circus* action that is defined in our link rules but not recursion, has the same effect on state and local variables, and the same behaviour as the action A. In other words, the semantics is preserved.

Proof. According to Lemma C.1.1 to C.1.27, the memory model of the link process from each action which is defined in our link rules preserves the semantics except recursion. Therefore, we conclude that the memory model of linked actions preserves the semantics of the actions.

□

C.2 F_{Ren}

Theorem C.2.1. The $F_{Ren}(A, \{(v_{old}, v_{new})\})$ function preserves the behaviour of A and has the same effect on variables v_{new} as on v_{old}.

Proof. Because F_{Ren} is just a syntactical replacement of variables v_{old} in A, it preserves the behaviour of A and has the same effect on variables v_{new} as on v_{old}. □

Lemma C.2.2 (F_{Ren} Transitivity)**.**

$$\begin{aligned} & F_{Ren}\left(F_{Ren}\left(t, \{(v_{old}, v_{mid})\}\right), \{(v_{mid}, v_{new})\}\right) \\ = & F_{Ren}\left(t, \{(v_{old}, v_{new})\}\right) \end{aligned}$$

C.3 External Choice Elimination

Lemma C.3.1. *Provided the type of x is a set T_x which has n elements $x_1, \cdots, x_n$, and P_i is a process. The construct below is not syntactically correct in Circus. It is introduced only for intermediate usage.*

$$\left(\begin{array}{ll} & (x == x_1) \mathbin{\&} P_1 \\ \square & (x == x_2) \mathbin{\&} P_2 \\ \square & \cdots \\ \square & (x == x_n) \mathbin{\&} P_n \end{array}\right)$$

, if x is x_i, it is simplified to P_i.

Appendix D

Circus Expressions, Operators and Predicates to B and CSP$_M$

In this Appendix D, the map from *Circus* expressions, operators, and predicates to their corresponding counterparts in CSP$_M$ are listed. They are categorized into eight groups: type, logic, basic and boolean, number, set, relation, function and sequence as shown in Section D.2. In the beginning of this chapter, a list of notations is presented in Section D.1 and will be used later in the map tables.

It is worth noting that the data structures in *Circus* can be very abstract. However the data structures in CSP are comparatively concrete. For instance, a given set in *Circus* only gives the name of the set and does not include the details about its elements. But this is not the case in CSP. Therefore, a configuration named $MAXINS$ is introduced to indicate how many elements the new set would have when concretising the given set in CSP. Furthermore, for some expressions, operators and predicates it is easy to find their counterparts in CSP$_M$ such as $\cup$ (`union` in CSP$_M$). But for others like $\nrightarrow$, we have to implement a corresponding one in CSP$_M$. All these new functions are provided as CSP libraries displayed in Section D.3.

D.1 Notations

See Table D.1.

D.2 Map of *Circus* Types, Expressions, Operators and Predicates to CSP

In this section, a map of *Circus* types, expressions, operators, and predicates to CSP$_M$ is listed. For some of them, there are direct counterparts in CSP$_M$. But for others, they are not the case. Therefore, we define these functions in CSP$_M$ as a library and include them by `include lib_file.csp` as required. All libraries defined are included in Section D.3.

D.2.1 Type

D.2.1.1 Free Type

Constants Only

$$FT ::= c1 \mid \ldots \mid cn$$

is transformed to

```
datatype FT = c1 | ... | cn
```

Constants and Constructors without Recursion

$$FT ::= c1 \mid \cdots \mid cm \mid d1\langle\langle E1\rangle\rangle \mid \cdots \mid dn\langle\langle En\rangle\rangle$$

is transformed to

```
datatype FT = c1 | ... | cm | d1.E1c | ... | dn.Enc
```

Then the instance $di(e)$ is translated to `di.ec`.
For example,

$$Status ::= ok \mid err\langle\langle 1\,..\,5\rangle\rangle$$

is transformed to

```
datatype Status = ok | err.{1..5}
```

Additionally, $err(1)$ is translated to `err.1` in CSP.

Constants and Constructors with Recursion

$$FT ::= c1 \mid \cdots \mid cm \mid d1\langle\langle E1[FT]\rangle\rangle \mid \cdots \mid dn\langle\langle En[FT]\rangle\rangle$$

is transformed to

```
datatype FT = c1 | ... | cm | d1.E1c | ... | dn.Enc
```

For instance [63, Example 10.4],

$$Tree ::= leaf\langle\langle \mathbb{N}\rangle\rangle \mid Branch\langle\langle Tree \times Tree\rangle\rangle$$

is transformed to

Table D.1: Notations

Circus	CSP	Description
x, y	`x,y`	General variables
T	`Tc`	Type of variable
P	`Pc`	Logic formula
Q	`Qc`	Logic formula
p	`pc`	Predicate
$p[y/x]$	`pc[y/x]`	A replacement of each occurrence of x by y
$e, e1, e2$	`ec, e1c, e2c`	Expression
Te	`Tec`	Type of expression e
$s, s1, s2$	`s, s1, s2`	Set variables
t	`t`	Tuple variable
$S, S1, S2$	`Sc, S1c, S2c`	Set
SS	`SSc`	Set over Set
$r, r1, r2$	`r, r1, r2`	Relation variables
$f, f1, f2$	`f, f1, f2`	Function variables
$sq, sq1, sq2$	`sq, sq1, sq2`	Sequence variables
$sqsq$	`sqsq`	Sequence over sequence variable
sqs	`sqs`	Sequence over set variable
$sch, sch1, sch2$	`schc, sch1c, sch2c`	Schema variable/reference
se	`sec`	Schema expression

```
datatype Tree = leaf.Nat | branch.(Tree, Tree)
```

Additionally, $branch(branch(leaf\,3, leaf\,5), leaf\,9)$ is translated to

```
branch.(branch.((leaf.3), (leaf.5)), (leaf.9))
```

in CSP.

In order to support unification of free types between CSP and Z, we modified the `specfile.pl` file in ProB, especially the `type_ok2` and `check_type` predicates, to make free types compatible between CSP and Z. This change is an implementation of our solution to treat free types in CSP and it is not implemented in the general ProB.

D.2.1.2 Schema

Schema as Type A schema as type is linked to a set comprehension of an ordered pair. And for each element in the pair it is given a tag name to identify its field in the schema. For example, Sch_rec_x means the field x in the schema Sch.

$$\begin{aligned} & \Phi(Sch == [\, x : Tx \,;\, y : Ty \mid pred(x,y) \,]) \\ = \quad & Sch = \{(Sch\_rec\_x.x, Sch\_rec\_y.y) \mid x \leftarrow \Phi(Tx), y \leftarrow \Phi(Ty), \Phi(pred(x,y))\} \end{aligned}$$

The field tag is defined as a data type.

$$\text{datatype } Sch\_rec\_field = Sch\_rec\_x.\Phi(Tx) \mid Sch\_rec\_y.\Phi(Ty)$$

And the selection operation of the object with the schema type is linked as well.

$$\Phi(obj.x) = select\_field\_Sch(obj, Sch\_rec\_x)$$

where the $select\_field\_Sch$ is a function defined in CSP to get the value of a specified field.

$$\begin{aligned} & select\_field\_Sch((Sch\_rec\_x.x, \_), Sch\_rec\_x) = x \\ & select\_field\_Sch((\_, Sch\_rec\_y.y), Sch\_rec\_y) = y \end{aligned}$$

By this conversion, a schema is translated to a set of ordered pairs. At the same time, we modified the implementation of ProB to identify the element with a tag having the $\_rec\_$ pattern as an element in a schema, and its pair $(Sch\_rec\_x.x,\ Sch\_rec\_y.y)$ as a binding of the schema $\langle\!| x \leadsto x, y \leadsto y |\!\rangle$. This modification is to implement our solution of schema types support in CSP and it is not supported in the general ProB as well, which is similar to the change made for free types. Finally, a variable with the Sch type in Z can synchronise with a variable with the Sch type in CSP. The translation of $PumpCtr$ is an example.

For instance, $PumpCtr$ is normalised and finally linked to the implementation below in CSP. Note that the prefix $Analyser$ is the name of the process.

```
datatype PumpCtr_rec_field = PumpCtr_rec_pa_1.{0, P}
    | PumpCtr_rec_pa_2.{0, P} | PumpCtr_rec_pcst.PCState
    | PumpCtr_rec_pst.PState
PumpCtr = {(PumpCtr_rec_pa_1.pa_1, PumpCtr_rec_pa_2.pa_2,
    PumpCtr_rec_pcst.pcst, PumpCtr_rec_pst.pst) | pa_1 <- {0, P},
    pa_2 <- {0, P}, pcst <- PCState, pst <- PState, ...}
select_field_PumpCtr((PumpCtr_rec_pa_1.x, _, _, _),
    PumpCtr_rec_pa_1) = x
select_field_PumpCtr((_, PumpCtr_rec_pa_2.x, _, _),
    PumpCtr_rec_pa_2) = x
select_field_PumpCtr((_, _, PumpCtr_rec_pcst.x, _),
    PumpCtr_rec_pcst) = x
select_field_PumpCtr((_, _, _, PumpCtr_rec_pst.x),
    PumpCtr_rec_pst) = x
```

And the selection $(pumpctr).pst$ is translated to

```
select_field_PumpCtr(pumpctr, PumpCtr_rec_pst)
```

Schema as Predicate Schemas are frequently used as predicates in the behavioural part, such as $(vzero.RateZero \rightarrow \mathbf{Skip})$, where $RateZero$ is a schema, in the $Analyser$ process.

$$RateZero == [\, VSensor \mid va_1 = 0 \land va_2 = 0\,]$$

For a schema as predicate, we assume all variables in the schema are declared and the schema is normalised, and therefore it is equal to this schema's predicate, such as $\Phi(SchAsPred) = \Phi(pred)$.

D.2.2 Logic

See Table D.2.

Table D.2: Logic

Operator	*Circus* Symbol	*Circus* LaTeX	CSP$_M$	Description
Logical and	$P \land Q$	`P \land Q`	`Pc and Qc`	
Logical or	$P \lor Q$	`P \lor Q`	`Pc or Qc`	
Logical implication	$P \Rightarrow Q$	`P \implies Q`	`implies(Pc, Qc)`	`implies(x, y) = not x or y`
Logical equivalence	$P \Leftrightarrow Q$	`P \iff Q`	`iff(Pc, Qc)`	`iff(x, y) = implies(x,y) and implies(y,x)`
Logical negation	$\neg P$	`\lnot P`	`not Pc`	
Universal quantification	$\forall x : T \bullet p$	`\forall x:T \spot p`	`forall(Tc, (\ x @ pc))`	
Existential quantification	$\exists x : T \bullet p$	`\exists x:T \spot p`	`exists(Tc, (\ x @ pc))`	
Unique existence	$\exists_1 x : T \bullet p$	`\exists_1 x:T \spot p`	`exists_1(Tc, (\ x @ pc))`	

D.2.3 Basic and Boolean

See Table D.3.

D.2.3.1 lambda

$\lambda x : Tx ; y : Ty \bullet e$ is mapped to

```
\ x,y @ if member((x,y), {x,y | x<-Txc, y<-Tyc})
        then ec
        else undefined({})
```

$\lambda\, x : Tx \,;\, y : Ty \mid p \bullet e$ is mapped to

```
\ x,y @ if member((x,y), {vx,vy | vx<-Txc, vy<-Tyc, p(vx,vy)})
        then ec
        else undefined({})
```

D.2.3.2 mu

If the existence and uniqueness properties of mu-expression does not hold, then the `mu` function defined below will trigger an error during model checking because `pick` function is only defined for a set with exactly one element.

```
mu(s, P, e) =
    let
        pick({x}) = x
        r = pick({e(x) | x <- s, P(x)})
    within r
```

Table D.3: Basic and Boolean

Operator	*Circus* Symbol	*Circus* LaTeX	CSP$_M$	Description
Equality	$e1 = e2$	`e1 = e2`	`e1c == e2c`	
Inequality	$e1 \neq e2$	`e1 \neq e2`	`e1c != e2c`	
Conditional	**if** p **then** $e1$ **else** $e2$	`\IF p \THEN e1 \ELSE e2`	`if pc then e1c else e2c`	
Lambda-expression	$\lambda\, x : T \mid p \bullet e$	`\lambda x:T | p @ e`	`\ x @ ec`	Section D.2.3.1.
Mu-expression	$\mu\, x : T \mid p \bullet e$	`\mu x:T | p @ e`	`mu(Tc, pc, ec)`	
True	**True**	`\true`	`true`	
False	**False**	`\false`	`false`	
Boolean	$\mathbb{B}$	`\boolean`	`Bool`	

D.2.4 Number

See Table D.4.

D.2.5 Set

See Table D.5.

D.2.6 Relation

See Table D.6.

D.2.7 Function

See Table D.7.

D.2.8 Sequence

See Table D.8.

D.3 CSP_M Library Functions

The libraries are also available either in accompanying CD-ROM or online at GitHub [104].

D.3.1 lib_basic.csp

```
-- lib_basic.csp

-- lambda
undefined({}) = {}

-- mu
-- \mu x:T | p @ e
-- there exists exactly one x which makes p hold, and return the expression
mu(s, P, e) =
    let
       pick({x}) = x
       r = pick({e(x) | x <- s, P(x)})
    within r
```

D.3.2 lib_num.csp

```
-- natural number
Nat={0..MAXINT}
Nat1={1..MAXINT}

-- succ
```

Table D.4: Number

Operator	*Circus*		CSP_M	Description
	Symbol	LaTeX		
Natural numbers	$\mathbb{N}$	`\nat`	`Nat`	`Nat={0..MAXINT}`
Strictly positive integers	$\mathbb{N}_1$	`\nat_1`	`Nat_1`	`Nat_1={1..MAXINT}`
Successor	$succ\ x$	`succ~x`	`succ(x)`	`succ(x)=x+1`
Integers	$\mathbb{Z}$	`\num`	`Int`	
Non empty integers	$\mathbb{Z}_1$	`\num_1`	`num_1`	
Negation	$-$	`\negate`	`-`	
Subtraction	$-$	`-`	`-`	
Summation	$+$	`+`	`+`	
Multiplication	$*$	`*`	`*`	
Integer devision	**div**	`\div`	`/`	
Integer modulus	mod	`\mod`	`%`	
Less than equal	$\leq$	`\leq`	`<=`	
Less than	$<$	`<`	`<`	
Greater than equal	$\geq$	`\geq`	`>=`	
Greater than	$>$	`>`	`>`	
Minimum	$min\ s$	`min~s`	`min(s)`	
Maximum	$max\ s$	`max~s`	`max(s)`	

Table D.5: Set

Operator	Circus		CSP$_M$	Description
	Symbol	LaTeX		
Number range	$x..y$	`x \upto y`	`{x .. y}`	
Given set	$[NAME]$	`[NAME]`	`datatype NAME = NAME_GS.{0..MAXINS}`	
Set enumeration	$\{x, y, dots\}$	`{x, y, ...}`	`{x, y, ...}`	
Set comprehension	$\{x : T \mid p \bullet e\}$	`{ x:T | p @ e }`	`{ec | x<-Tc, pc}`	
Characteristic set comprehension	$\{x : T \mid p\}$	`{ x:T | p }`	`{x | x<-Tc, pc}`	
Simple set comprehension	$\{x : T \bullet e\}$	`{ x:T @ e }`	`{ec | x<-Tc}`	
Set membership	$x \in S$	`x \in S`	`member(x, Sc)`	
Not in	$x \notin S$	`x \notin S`	`not member(x, Sc)`	
Cartesian product	$S1 \times S2$	`S1 \cross S2`	`cross(S1c, S2c)`	
Cartesian product	$S1 \times S2 \times \cdots \times Sn$	`S1 \cross S2 \cross ... \cross Sn`	`(S1c, S2c, ..., Snc)`	
Empty Set	$\varnothing$	`\emptyset`	`{}`	
Subset	$S1 \subseteq S2$	`S1 \subseteq S2`	`leq(S1c, S2c)`	
Proper subset	$S1 \subset S2$	`S1 \subset S2`	`le(S1c, S2c)`	
Power sets	$\mathbf{P}\ S$	`\power~S`	`Set(Sc)`	
Non-empty subsets	$\mathbf{P}_1\ S$	`\power_1~S`	`diff(Set(Sc),{{}})`	
Set union	$S1 \cup S2$	`S1 \cup S2`	`union(S1c, S2c)`	
Set intersection	$S1 \cap S2$	`S1 \cap S2`	`inter(S1c, S2c)`	
Set difference	$S1 \setminus S2$	`S1 \setminus S2`	`diff(S1c, S2c)`	
Set symmetric difference	$S1 \ominus S2$	`S1 \symdiff S2`	`symdiff(S1c,S2c)`	
Generalised union	$\bigcup SS$	`\bigcup SS`	`Union(SSc)`	
Generalised intersection	$\bigcap SS$	`\bigcap SS`	`Inter(SSc)`	
Finite sets	$\mathbf{F}\ S$	`\finset S`	`Set(Sc)`	only consider finite set S.
Non empty finite sets	$\mathbf{F}_1\ S$	`\finset_1 S`	`finset_1`	
cardinality	$\#\ S$	`\#~S`	`len(Sc)`	

Table D.6: Relation

Operator	*Circus*		CSP$_M$	Description
	Symbol	LaTeX		
Relation	$S1 \leftrightarrow S2$	`S1 \rel S2`	`rel(S1c, S2c)`	
Maplet	$x \mapsto y$	`x \mapsto y`	`(x, y)`	
Domain	$\operatorname{dom} r$	`\dom r`	`dom(r)`	
Range	$\operatorname{ran} r$	`\ran r`	`ran(r)`	
Identity	$\operatorname{id} S$	`\id S`	`id(Sc)`	
Relational composition	$r1 \fatsemi r2$	`r1 \comp r2`	`comp(r1, r2)`	
Functional composition	$r1 \circ r2$	`r1 \circ r2`	`circ(r1,r2)`	
Domain restriction	$s \lhd r$	`s \dres r`	`dres(s, r)`	
Range restriction	$r \rhd s$	`r \rres s`	`rres(r, s)`	
Domain subtraction	$s ⩤ r$	`s \ndres r`	`ndres(s, r)`	
Range subtraction	$r ⩥ s$	`r \nrres s`	`nrres(r, s)`	
Relational inversion	$r^{\sim}$	`r\inv`	`inv(r)`	
Relational image	$r \llparenthesis s \rrparenthesis$	`r \limg s \rimg`	`img(s,r)`	
Overriding	$r1 \oplus r2$	`r1 \oplus r2`	`oplus(r1, r2)`	
First projection function	$\mathit{first}\ t$	`first~t`	`first(t)`	
Second projection function	$\mathit{second}\ t$	`second~t`	`second(t)`	
Iteration	$\mathit{iter}\ n\ r$	`iter~n~r`	`itern(r,n)`	
Iteration	$(r^{\ n})$	`(r~^{~n~})`	`itern(r,n)`	
Transitive closure	r^{+}	`r \plus`		Not implemented
Reflexive transitive closure	r^{*}	`r \star`		Not implemented

```
succ(n)=n+1

-- define them for num1
-- MAXINT=1000
-- MININT=-1000

-- num1
num1={x | x <- {MININT..MAXINT}, x != 0}

-- min and max
min(<x>) = x
min(<x>^s^<y>) = if x <= y then min(<x>^s) else min(<y>^s)

min(s) = min(seq(s))

max(<x>) = x
max(<x>^s^<y>) = if x >= y then max(<x>^s) else max(<y>^s)

max(s) = max(seq(s))
```

D.3.3 lib_card.csp

```
-- lib_card

-- calculate cardinality of set and sequence using the same function "len"
len(<>) = 0
len(<x>) = 1
```

Table D.7: Function

Operator	*Circus*		CSP$_M$	Description
	Symbol	LaTeX		
Function application	$f(x)$	`f(x)`	`fa(f, x)`	
Partial function	$S1 ⇸ S2$	`S1 \pfun S2`	`pfun(S1c, S2c)`	
Total function	$S1 \rightarrow S2$	`S1 \fun S2`	`tfun(S1c, S2c)`	
Partial injection	$S1 ⤔ S2$	`S1 \pinj S2`	`pifun(S1c, S2c)`	
Injection	$S1 \rightarrowtail S2$	`S1 \inj S2`	`tifun(S1c, S2c)`	
Partial surjection	$S1 ⤀ S2$	`S1 \psurj S2`	`psfun(S1c, S2c)`	
Surjection	$S1 \twoheadrightarrow S2$	`S1 \surj S2`	`tsfun(S1c, S2c)`	
Bijection	$S1 ⤖ S2$	`S1 \bij S2`	`bjfun(S1c, S2c)`	
Finite partial function	$S1 ⇻ S2$	`S1 \ffun S2`	`pfun(S1c, S2c)`	
Finite partial injection	$S1 ⤕ S2$	`S1 \finj S2`	`pifun(S1c, S2c)`	

Table D.8: Sequence

	Circus			
Operator	Symbol	LaTeX	CSP$_M$	Description
Cardinality	$\#\ sq$	`\#~sq`	`len(sq)`	
Finite seq	$seq\ S$	`\seq S`	`fseq(Sc)`	
Non empty seq	$seq_1\ S$	`\seq_1 S`	`seq1(Sc)`	
Injective seq	iseq S	`\iseq S`	`iseq(Sc)`	
Sequence brackets	$\langle , , \rangle$	`\langle \listarg \rangle`	`< , , >`	
Concatenation	$sq1 \frown sq2$	`sq1 \cat sq2`	`sq1^sq2`	
Reverse	$rev\ sq$	`rev~sq`	`reverse(sq)`	
Head	$head\ sq$	`head~sq`	`head(sq)`	
Last	$last\ sq$	`last~sq`	`last(sq)`	
Tail	$tail\ sq$	`tail~sq`	`tail(sq)`	
Front	$front\ sq$	`front~sq`	`front(sq)`	
Re-indexing	$squash\ sq$	`squash~sq`	`squash(sq)`	
Extraction	$S \upharpoonleft sq$	`S \extract sq`	`extract(Sc, sq)`	
Filtering	$sq \upharpoonright S$	`sq \filter S`	`filter(Sc, sq)`	
Prefix	$sq1 \leq sq2$	`sq1 \prefix sq2`	`prefix(sq1, sq2)`	
Suffix	$sq1 \geq sq2$	`sq1 \suffix sq2`	`suffix(sq1, sq2)`	
Infix	$sq1$ infix $sq2$	`sq1 \infix sq2`	`infix(sq1, sq2)`	
Distributed Concatenation	$\frown/\ sqsq$	`\dcat~sqsq`	`concat(sqsq)`	
Disjoint sets	disjoint sqs	`\disjoint~sqs`	`disjoint(sqs)`	
Set partitioning	sqspartition S	`sqs \partition S`	`partition(sqs, Sc)`	

```
len(<x>^s) = len(s) + 1
len(s) = len(seq(s))
```

D.3.4 lib_log.csp

```
-- lib_log.csp

-- implies
implies(p1, p2) = ((not p1) or p2)

-- iff
iff(p1, p2) = implies(p1, p2) and implies(p2, p1)

-- forall
-- example: forall x:N @ x + 1 > x    =>    forall(N, \ x @ x + 1 > x)
-- there doesn't exist a x from s such that P(x) is false
forall(s,P) = empty({x | x<-s, not P(x)})

-- exist
-- example: exist x:N @ x*x=25
-- there does exist at least one x from s such that P(x) is true
exists(s,P) = not empty({x | x<-s, P(x)})

-- exist_1
exists_1(s,P) = card({x | x<-s, P(x)}) == 1
```

D.3.5 lib_set.csp

```
-- lib_set.csp

-- MAXINS for given set, should be defined in the beginning of
--  specification
-- MAXINS = 3

-- compare of two sets: equal(s1,s2) or "=="
equal(s1,s2) = empty({x | x <- s1, not member(x, s2)}) and
               empty({x | x <- s2, not member(x, s1)})

-- compare of two sets: less than or equal
-- CSPM (<=) can compare the subset but not supported in ProB by now
leq(s1,s2)=empty({x | x <- s1, not member(x,s2)})

-- compare of two sets: less than or equal
-- CSPM (<) can compare the subset but not supported in ProB by now
le(s1,s2)=leq(s1,s2) and not empty({x | x <- s2, not member(x, s1)})

-- \power_1
power_1(s) = diff(Set(s), {{}})

-- \symdiff
symdiff(s1, s2) = union(diff(s1,s2), diff(s2,s1))
```

```
-- Cartesian Product
cross(X, Y) = {(x,y) | x <- X, y <- Y}

-- finset_1
finset_1(s) = {x | x <- s, x != {}}
```

D.3.6 lib_rel.csp

```
-- lib_rel.csp

--rel(X, Y) = { (x,y) | x<-X, y<-Y }
rel(X, Y) = Set(cross(X, Y))

-- id
id(s) = {(x,x) | x <- s}

-- first and second
first((x,y)) = x
second((x,y)) = y

-- domain and range
dom(s) = { x | (x,y)<-s }
ran(s) = { y | (x,y)<-s }

-- comp
comp(s1,s2) = {(x,v) | (x,y)<-s1, (u,v)<-s2, y==u}

-- circ (functional composition or backward relation composition)
circ(s1,s2) = comp(s2,s1)

-- domain restriction
dres(rs,s) = {(x,y) | (x,y)<-s, member(x,rs)}

-- range restriction
rres(s,rs) = {(x,y) | (x,y)<-s, member(y,rs)}

-- domain subtraction
ndres(rs,s) = {(x,y) | (x,y)<-s, not member(x,rs)}

-- range subtraction
nrres(s,rs) = {(x,y) | (x,y)<-s, not member(y,rs)}

-- inv
inv(s) = {(y,x) | (x,y)<-s}

-- image
-- relational image
--img(A,s) = ran(dres(A,s))
img(s,r) = {y | (x,y)<-r, member(x,s)}

-- overriding
oplus(s1,s2) = union(
        {(x,y) | (x,y) <- s1, not member(x,dom(s2))},
        {(u,v) | (u,v) <- s2, member(u,dom(s1))} )

-- iter (s is a homogeneous relation X <--> X -
--       source and target have the same type)
iter(s,0) = id(dom(s))
iter(s,1) = s
iter(s,n) = comp(s,iter(s,n-1))
-- negative number is not supported in pattern match
-- iter(s, -1) = iter(inv(s), 1)
-- iter(s,n) = comp(inv(s),iter(s,n+1))
itern(s,n) = if n < 0 then iter(inv(s), 0 - n) else iter(s,n)
-- iter(s,n) = if n < 0 then comp(inv(s), iter(s, n+1)) else iterr(s,n)

-- _+ and _*: transitive closure and reflexive-transitive closure
iters(s,1) = iter(s,1)
iters(s,n) = Union({iters(s,n-1), iter(s,n)})
star(s) = iters(s,10)

--iterp(s,0) = iter(s,0)
--iterp(s,1) = Union({iterp(s,0), iter(s,1)})
--iterp(s,n) = Union({iterp(s,n-1), iter(s,n)})
--plus(s) = iterp(s,10)
plus(s) = union(star(s), iter(s,0))
```

D.3.7 lib_fun.csp

```
-- lib_fun.csp

-- function means there's no element which can map to two different values
fun(X, Y) = { s | s<-rel(X, Y),
    empty({x1 | (x1,y1)<-s, (x2,y2)<-s, x1 == x2 and y1 != y2})}

-- partial function: not every element in X has the corresponding value
--  in Y
-- implement 1: by its definition
-- pfun(X, Y) = { s | s<-rel(X,Y),
--   empty({x1 | (x1,y1)<-s, (x2,y2)<-s, x1 == x2 and y1 != y2})}

-- implement 2: efficient
---------------------------- pfun [START] --------------------------------
-- a partial function pfun(X, Y) finally is a set of (set of pairs)
-- {{(x1,y1), (x2, y2) ... }, {(x1, y2)...}, {()}}
-- in its subset, the source of each pair (X) shall not be unique,
--      therefore {(x1, y1), (x1, y2)} is not allowed
-- however the destination of each pair (y) don't have to be unique,
--      therefore {(x1, y1), (x2, y2)} is valid
--
-- the algorithms to calculate a partial function of X and Y, shown below
-- 1. generally it is a union of
--  a) taking zero item from X, => {}
--  b) taking one item from X, => {{(x1,y1)}, {(x1,y2)},...{(x2,y1)}, ...}
--  c) taking two items from X, => {{(x1,y1), (x2, y1)}, ...}
--  i) ...
--  n) taking n items from X, => {{(x1,y1), (x2, y1), ..., (xn, y1)}, ...}
-- 2. for each small step a), b), ..., n), use pcomb(X, Y, n)
--  pcomb(X, Y, n) is used to compute all possible pairs which size is n.
--  for example, pcomp({1,2},{3},1) gets {{(1,3)},{(2,3)}}
--    and the cardinality is one
-- 3. for pcomp,
--  a) at first, calculate all possible subset of X and its cardinality
--     is n (by select2)
--     for example, select2({1,2,3}, 2) = {<1,2>,<1,3>,<2,3>}
--  b) then calculate all possible combination of elements from Y and
--     its size is n as well.
--     for example, select1({1,2}, 2) = {<1,1>, <1,2>, <2,1>, <2,2>}
--  c) join the sequences from select2 and select1 together to get pcomp

-- select n items from x, order matters and duplicate items are allowed
-- select1({1,2}, 2) = {<1,1>, <1,2>, <2,1>, <2,2>}
select1(Y, 0) = { <> }
select1(Y, 1) = { <y> | y <- Y}
select1(Y, n) = { <y>^ss | y <- Y, ss <- select1(Y, n-1) }

-- select n items from x, order doesn't matter and duplicate items are
-- not allowed
-- select({1,2}, 1) = {<1>,<2>}
-- select({1,2}, 2) = {<1,2>}
select2_1(X, 0) = {}
-- choose more than 1 from <x> leads to emptyset
select2_1(<x>, n) = if n > 1 then {} else { <x> }
select2_1(X, 1) = { <x> | x <- set(X) }
select2_1(<x>^s, n) = union({ <x> ^ ss | ss <- select2_1(s, n-1)},
                        select2_1(s, n))

select2(X, n) = select2_1(seq(X), n)

-- join(s1, s2) to form a {(s11, s21), (s12, s22), ..., (s1n, s2n)}
-- s1 and s2 are the same size sequences
-- s1 = <s11, s12, ..., s1n>
-- s2 = <s21, s22, ..., s2n>
pjoin(<x>, <y>) = {(x,y)}
pjoin(<x>^s1, <y>^s2) = union({(x,y)}, pjoin(s1, s2))

--
pcomb(X, Y, n) = { pjoin(s1, s2) | s1 <- select2(X, n),
                    s2 <- select1(Y, n)}

pfun1(X, Y, 1) = {{(x,y)} | x <- X, y <- Y}
pfun1(X, Y, n) = union(pfun1(X, Y, n-1), pcomb(X, Y, n))

pfun(X, Y) = union(pfun1(X, Y, card(X)), {{}})
------------------------------ pfun [END] ---------------------------------
```

```
-- total function: every element in X has the corresponding value in Y
-- implementation 1:
-- tfun(X, Y) = { s | s<-pfun(X, Y), dom(s) == X}

-- implementation 2:
tfun1(<>, Y) = {}
--tfun(X, {}) = {}
tfun1(<x>, Y) = {{(x, y)} | y <- Y}
tfun1(<x>^s, Y) = { union(sx, ss) | sx <- tfun1(<x>, Y), ss <- tfun1(s, Y)}

tfun(X, Y) = tfun1(seq(X), Y)

-- Partial injections: every value in Y is mapped to
-- up to one element in X
pifun(X, Y) = { s | s<-pfun(X, Y),
    empty({x1 | (x1,y1)<-s, (x2,y2)<-s, y1 == y2 and x1 != x2})}

-- Total injections: Partial injections and total function
tifun(X, Y) = inter(pifun(X, Y), tfun(X, Y))

-- Partial surjections: every value in Y is mapped to
-- at least one element in X
psfun(X, Y) = { s | s<-pfun(X, Y), ran(s) == Y}

-- Total surjections: Partial surjections and total function
tsfun(X, Y) = inter(psfun(X, Y), tfun(X, Y))

-- Bijections: total surjections and total injections
bjfun(X, Y) = inter(tsfun(X, Y), tifun(X, Y))

-- function application: f(x)
-- fa(f, x)
-- fa(f)(x) is not supported in ProB
fa(f, a) =
    let
        pick({x}) = x
        y = pick({yy | (x, yy)<-f, x == a})
    within y
```

D.3.8 lib_seq.csp

```
-- lib_seq.csp

-- reverse(s) = if null(s) then <> else reverse(tail(s)) ^ <head(s)>
reverse(<>) = <>
reverse(<x>^s) = reverse(s)^<x>

-- last
last(s) = if null(s) then <> else head(reverse(s))

-- front
front(s) = reverse(tail(reverse(s)))
```

```
-- Injective sequence
--unique(s) = < x | x<-s, y<-s, x==y>
unique(<>) = true
unique(<x>) = true
unique(<x>^s) = if not elem(x,s) then unique(s) else false

-- for seq
fseq(s) = {squash(ss) | ss <- pfun({1..MAXINS}, s) }

-- seq_1
seq1(s) = diff(Seq(s), {<>})

-- iseq: injective sequence. No duplicate elements in the sequence
-- perm(s): calculate all permutation of set s
-- perm({1,2,3}) = {<1,2,3>, <1,3,2>, <2,1,3>, <2,3,1>, <3,1,2>, <3,2,1>}
perm({}) = {<>}
perm({x}) = {<x>}
perm(s) = { <x>^z | x <- s, y <- Set(s), y == diff(s, {x}), z <- perm(y)}

iseq(s) = { y | ss <- Set(s), y <- perm(ss)}

-- squash
-- such as Z:squash({1 |-> a, 5 |-> b, 3 |-> c}) = <a, c, b>
-- such as csp:squash({(1,a), (5,b),(3,c)}) = <a, c, b>
-- limitation: the elements in s cannot have the same x value.
--    For example, for {(1,a), (1,b)}, squash will fail
squash(s) =
  let
    pick({x}) = x
    -- how many items which 1st item x are below b
    below(b) = card({ x | (x,y)<-s, x <= b })
    -- get all pairs, such as {(a,1),(b,3),(c,2)}
    pairs    = { (y, below(x)) | (x,y)<-s }
    -- return the y value of a x below which there are i number of items
    select(i)= pick({ y | (y,n)<-pairs, i==n })
  within < select(i) | i <- <1..card(s)> >

-- A is a subset of domain of s (sequence)
--extract(A,s)
extract(A, s) = squash(dres(A, s))

-- filter(s,A)
filter(s, A) = squash(rres(A, s))

-- prefix
-- prefix
prefix(<>,<>) = true
prefix(<>,t) = true
prefix(<x>^s, <>) = false
prefix(<x>^s, <y>^t) = if x==y then prefix(s,t) else false
```

```
-- suffix
suffix(<>,<>) = true
suffix(<>,t) = true
suffix(s^<x>, <>) = false
suffix(s^<x>, t^<y>) = if x==y then suffix(s,t) else false

-- infix(s,t)
infix(_, <>) = false -- tested
infix(s, <x>) = if s==<x> then true else false
infix(s, <x>^t) = if prefix(s,<x>^t) then true else infix(s,t)

-- distributed concatenation
-- distributed concatenation
-- \dcat(s) and s is a sequence of sequence
dconcat(<>) = <>
dconcat(<s>) = s
dconcat(<x>^s) = x^dconcat(s)
dconcat(<x>^s^<y>) = x^dconcat(s)^y


-- disjoint(s)
--    - s is a sequence of set
disjoint(<>) = true
disjoint(<x>) = true
disjoint(<sx>^<sy>) = empty(inter(sx, sy))
disjoint(<sx>^ss^<sy>) =
    if disjoint(<sx>^<sy>) then
        disjoint(<sx>^ss) and disjoint(ss^<sy>)
    else false

-- partition(s, S)
--    - s is a sequence of set
--    - S is a set
partition(<>,{}) = true
partition(<sx>, C) =
    if equal(sx,C) then true
    else false
partition(<sx>^<sy>, C) =
    if disjoint(<sx>^<sy>) and equal(union(sx, sy), C) then true
    else false
partition(<sx>^ss^<sy>, C) =
    if disjoint(<sx>^ss^<sy>) and leq(union(sx, sy), C) then
        partition(ss, diff(C, union(sx, sy)))
    else false
```

Appendix E

Link from *Circus* to the combination of CSP and Z

Link Rules, which link constructs in *Circus* to the combination of CSP and Z, $CSP \parallel_B Z$, by applying individual rules defined in Chapter 4, are presented in this appendix.

E.1 Channel Declarations

E.1.1 Synchronisation Channel

Link Rule 1 (Synchronisation Channel).

$$\begin{aligned}
&\Upsilon\left(\mathbf{channel}\, c_1, \cdots, c_n\right) && \\
&= \Phi\left(R_{wrt}\left(\mathbf{channel}\, c_1, \cdots, c_n\right)\right) && [\text{Link Definition}]\\
&= \Phi\left(\mathbf{channel}\, c_1, \cdots, c_n\right) && [R_{wrt} \text{ Rule } 6]\\
&= \text{channel }\, c_1, \cdots, c_n && [\Phi \text{ Rule } 4]
\end{aligned}$$

□

E.1.2 Typed Channel

Link Rule 2 (Typed Channel).

$$\begin{aligned}
&\Upsilon\left(\mathbf{channel}\, c_1, \cdots, c_n : T\right) && \\
&= \Phi\left(R_{wrt}\left(\mathbf{channel}\, c_1, \cdots, c_n : T\right)\right) && [\text{Link Definition}]\\
&= \Phi\left(\mathbf{channel}\, c_1, \cdots, c_n : T\right) && [R_{wrt} \text{ Rule } 6]\\
&= \text{channel }\, c_1, \cdots, c_n : \Phi(T) && [\Phi \text{ Rule } 4]
\end{aligned}$$

□

E.1.3 Schema Typed Channel

Link Rule 3 (Schema Typed Channel).

$$\begin{aligned}
&\Upsilon\left(\mathbf{channelfrom}\ S\right) && \\
&= \Phi\left(R_{wrt}\left(\mathbf{channelfrom}\ S\right)\right) && [\text{Link Definition}]\\
&= \Phi\left(\begin{array}{l}\mathbf{channel}\ \ c_1, c_2, \ldots, c_n : T_c\\ \mathbf{channel}\ \ d_1, d_2, \ldots, d_m : T_d\end{array}\right) && [R_{wrt} \text{ Rule } 6]\\
&= \left\{\begin{array}{l}\text{channel }\ \ c_1, c_2, \ldots, c_n : \Phi(T_c)\\ \text{channel }\, d_1, d_2, \ldots, d_m : \Phi(T_d)\end{array}\right. && [\Phi \text{ Rule } 4]
\end{aligned}$$

provided

$$\begin{array}{|l}
\textit{S} \\ \hline
c_1, c_2, \ldots, c_n : T_c \\
d_1, d_2, \ldots, d_m : T_d \\ \hline
\end{array}$$

□

E.2 Channel Set Declarations

Link Rule 4 (Channel Set Declarations).

$$\begin{array}{ll}
\Upsilon\,(\textbf{channelset}\ N == \mathit{CSExp}) & \\
= \Phi\,(R_{wrt}\,(\textbf{channelset}\ N == \mathit{CSExp})) & [\text{Link Definition}] \\
= \Phi\,(\textbf{channelset}\ N == \mathit{CSExp}) & [R_{wrt}\ \text{Rule 7}] \\
= \mathrm{N} = \Phi(\mathit{CSExp}) & [\Phi\ \text{Rule 5}]
\end{array}$$

□

E.3 Channel Set Expressions

Link Rule 5 (Empty Channel Set).

$$\begin{array}{ll}
\Upsilon\,(\{\!|\ |\!\}) & \\
= \Phi\,(R_{wrt}\,(\{\!|\ |\!\})) & [\text{Link Definition}] \\
= \Phi\,(\{\!|\ |\!\}) & [R_{wrt}\ \text{Rule 8}] \\
= \{\!|\,|\!\} & [\Phi\ \text{Rule 6}]
\end{array}$$

□

Link Rule 6 (Channel Set Extension).

$$\begin{array}{ll}
\Upsilon\,(\{\!|\ c_1, c_2, \cdots, c_n\ |\!\}) & \\
= \Phi\,(R_{wrt}\,(\{\!|\ c_1, c_2, \cdots, c_n\ |\!\})) & [\text{Link Definition}] \\
= \Phi\,(\{\!|\ c_1, c_2, \cdots, c_n\ |\!\}) & [R_{wrt}\ \text{Rule 8}] \\
= \{\!|c_1, c_2, \cdots, c_n|\!\} & [\Phi\ \text{Rule 6}]
\end{array}$$

□

Link Rule 7 (Channel Set Reference and Expressions).

$$\begin{array}{ll}
\Upsilon\,(\mathit{CSRef}) & \\
= \Phi\,(R_{wrt}\,(\mathit{CSRef})) & [\text{Link Definition}] \\
= \Phi\,(\mathit{CSRef}) & [R_{wrt}\ \text{Rule 8}] \\
= \mathrm{CSRef} & [\Phi\ \text{Rule 6}]
\end{array}$$

In addition, channel set expressions by set union, set intersection, and set difference are linked similarly.

$$\begin{array}{ll}
\Upsilon\,(\mathit{CSExp}_1 \cup \mathit{CSExp}_2) & \\
= \Phi\,(R_{wrt}\,(\mathit{CSExp}_1 \cup \mathit{CSExp}_2)) & [\text{Link Definition}] \\
= \Phi\,(\mathit{CSExp}_1 \cup \mathit{CSExp}_2) & [R_{wrt}\ \text{Rule 8}] \\
= \mathrm{union}(\Phi(\mathit{CSExp}_1),\Phi(\mathit{CSExp}_2)) & [\Phi\ \text{Rule 6}]
\end{array}$$

$$\begin{array}{lr}
\Upsilon\,(\mathit{CSExp}_1 \cap \mathit{CSExp}_2) & \\
= \Phi\,(R_{wrt}\,(\mathit{CSExp}_1 \cap \mathit{CSExp}_2)) & \text{[Link Definition]} \\
= \Phi\,(\mathit{CSExp}_1 \cap \mathit{CSExp}_2) & [R_{wrt}\text{ Rule 8]} \\
= \mathrm{inter}(\Phi(\mathit{CSExp}_1),\Phi(\mathit{CSExp}_2)) & [\Phi\text{ Rule 6]}
\end{array}$$

$$\begin{array}{lr}
\Upsilon\,(\mathit{CSExp}_1 \setminus \mathit{CSExp}_2) & \\
= \Phi\,(R_{wrt}\,(\mathit{CSExp}_1 \setminus \mathit{CSExp}_2)) & \text{[Link Definition]} \\
= \Phi\,(\mathit{CSExp}_1 \setminus \mathit{CSExp}_2) & [R_{wrt}\text{ Rule 8]} \\
= \mathrm{diff}(\Phi(\mathit{CSExp}_1),\Phi(\mathit{CSExp}_2)) & [\Phi\text{ Rule 6]}
\end{array}$$

□

E.4 Explicitly Defined Processes

Link Rule 8 (Single Explicitly Defined Process).

$$\Upsilon\left(\begin{array}{l}
\textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{StPar} == [\, s_1 : T_1\, ; \cdots s_n : T_n \mid p\,] \\
\quad \mathit{Init} == [\, (\mathit{StPar})' \mid pi\,] \\
\quad \mathit{Pars} == [\cdots] \\
\quad \bullet\ A \\
\textbf{end}
\end{array}\right)$$

$$= \left\{\begin{array}{ll}
\Omega\left(R_{wrt}\left(\ \textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \cdots \textbf{end}\ \right)\right) & \text{State Part} \\
\\
\Phi\left(R_{wrt}\left(\ \textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \cdots \textbf{end}\ \right)\right) & \text{Behaviour Part}
\end{array}\right.$$

[Link Definition]

$$= \left\{\begin{array}{l}
\Omega\left(R_{wrt}\left(\begin{array}{l}
\textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{StPar} == [\, s_1 : T_1\, ; \cdots s_n : T_n \mid p\,] \\
\quad \mathit{Init} == [\, (\mathit{StPar})' \mid pi\,] \\
\quad \mathit{Pars} == [\cdots] \\
\quad \mathit{Op\_s}_1 == [\, \Xi \mathit{StPar}\, ; s_1! : T_1 \mid s_1! = s_1\,] \\
\quad \cdots \\
\quad \mathit{Op\_s}_n == [\, \Xi \mathit{StPar}\, ; s_n! : T_n \mid s_n! = s_n\,] \\
\quad \bullet\ A \\
\textbf{end}
\end{array}\right)\right) \\
\\
\Phi\left(R_{wrt}\left(\begin{array}{l}
\textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{StPar} == [\, s_1 : T_1\, ; \cdots s_n : T_n \mid p\,] \\
\quad \mathit{Init} == [\, (\mathit{StPar})' \mid pi\,] \\
\quad \mathit{Pars} == [\cdots] \\
\quad \mathit{Op\_s}_1 == [\, \Xi \mathit{StPar}\, ; s_1! : T_1 \mid s_1! = s_1\,] \\
\quad \cdots \\
\quad \mathit{Op\_s}_n == [\, \Xi \mathit{StPar}\, ; s_n! : T_n \mid s_n! = s_n\,] \\
\quad \bullet\ A \\
\textbf{end}
\end{array}\right)\right)
\end{array}\right.$$

[Additional Schemas R_{wrt} Rule 23]

$$= \begin{cases} \Omega \left(\begin{array}{l} \textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\ \quad \textbf{state}\, P\_StPar == [\, P\_s_1 : T_1 \,;\cdots P\_s_n : T_n \mid p \,] \\ \quad P\_Init == [\,(P\_StPar)' \mid pi\,] \\ \quad P\_Pars == [\cdots] \\ \quad P\_Op\_s_1 == [\,\Xi P\_StPar\,;\, s_1! : T_1 \mid s_1! = P\_s_1\,] \\ \quad \ldots \\ \quad P\_Op\_s_n == [\,\Xi P\_StPar\,;\, s_n! : T_n \mid s_n! = P\_s_n\,] \\ \quad \bullet\ R_{wrt}(A) \\ \textbf{end} \end{array} \right) \\ \\ \Phi \left(\begin{array}{l} \textbf{process}\ \ P \mathrel{\widehat=} \textbf{begin} \\ \quad \textbf{state}\, P\_StPar == [\, P\_s_1 : T_1 \,;\cdots P\_s_n : T_n \mid p \,] \\ \quad P\_Init == [\,(P\_StPar)' \mid pi\,] \\ \quad P\_Pars == [\cdots] \\ \quad P\_Op\_s_1 == [\,\Xi P\_StPar\,;\, s_1! : T_1 \mid s_1! = P\_s_1\,] \\ \quad \ldots \\ \quad P\_Op\_s_n == [\,\Xi P\_StPar\,;\, s_n! : T_n \mid s_n! = P\_s_n\,] \\ \quad \bullet\ R_{wrt}(A) \\ \textbf{end} \end{array} \right) \end{cases}$$

[Renaming R_{wrt} Rule 24]

$$= \begin{cases} \Omega_3 \left(\Omega_2 \left(\begin{array}{l} P\_StPar == [\, P\_s_1 : T_1 \,;\cdots P\_s_n : T_n \mid p \,] \\ State == P\_StPar \\ Init == [\,(State)' \mid pi\,] \\ P\_Pars == [P\_Pars.decl \mid P\_Pars.p] \\ P\_Op\_s_1 == [\,\Xi P\_StPar\,;\, s_1! : T_1 \mid s_1! = P\_s_1\,] \\ \ldots \\ P\_Op\_s_n == [\,\Xi P\_StPar\,;\, s_n! : T_n \mid s_n! = P\_s_n\,] \end{array} \right) \right) \\ \\ \mathrm{P} {=} \Phi\,(R_{wrt}(A)) \end{cases}$$

[Ω_1 Rule 1 and Φ Rule 8]

$$= \begin{cases} \Omega_3 \left(\begin{array}{l} P\_StPar \mathrel{\widehat=} [\, P\_s_1 : T_1 \,;\cdots P\_s_n : T_n \mid p \,] \\ State \mathrel{\widehat=} P\_StPar \\ Init \mathrel{\widehat=} [\, State' \mid pi\,] \\ P\_Pars \mathrel{\widehat=} [P\_Pars.decl \mid P\_Pars.p] \\ P\_Op\_s_1 \mathrel{\widehat=} [\,\Xi P\_StPar\,;\, s_1! : T_1 \mid s_1! = P\_s_1\,] \\ \ldots \\ P\_Op\_s_n \mathrel{\widehat=} [\,\Xi P\_StPar\,;\, s_n! : T_n \mid s_n! = P\_s_n\,] \end{array} \right) \\ \\ \mathrm{P} {=} \Phi\,(R_{wrt}(A)) \end{cases}$$

[Ω_2 Rule 4 and Ω_2 Rule 5]

□

Link Rule 9 (Explicitly Defined Processes). Provided there are n explicitly defined processes, then they are linked below.

$$
\begin{array}{l}
\Upsilon \left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{process}\ \ P_1 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, \mathit{StPar} == [\, s_{11} : T_{11} \,; \cdots s_{1m_1} : T_{1m_1} \mid ps_1 \,] \\
\quad \mathit{Init} == [\, (\mathit{StPar})' \mid pi_1 \,] \\
\quad \mathit{Pars} == [\, decl_1 \mid p_1 \,] \\
\quad \bullet\ A \\
\textbf{end}
\end{array} \right) \\
\cdots \\
\left(\begin{array}{l}
\textbf{process}\ \ P_n \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\, \mathit{StPar} == [\, s_{n1} : T_{n1} \,; \cdots s_{nm_n} : T_{nm_n} \mid ps_n \,] \\
\quad \mathit{Init} == [\, (\mathit{StPar})' \mid pi_n \,] \\
\quad \mathit{Pars} == [\, decl_n \mid p_n \,] \\
\quad \bullet\ A \\
\textbf{end}
\end{array} \right)
\end{array} \right) \\
= \left\{ \begin{array}{ll}
\Omega \left(R_{wrt} \left(\begin{array}{l}
\textbf{process}\ \ P_1 \mathrel{\widehat=} \textbf{begin} \cdots \textbf{end} \\
\cdots \\
\textbf{process}\ \ P_n \mathrel{\widehat=} \textbf{begin} \cdots \textbf{end}
\end{array} \right) \right) & \text{State Part} \\
\\
\Phi \left(R_{wrt} \left(\begin{array}{l}
\textbf{process}\ \ P_1 \mathrel{\widehat=} \textbf{begin} \cdots \textbf{end} \\
\cdots \\
\textbf{process}\ \ P_n \mathrel{\widehat=} \textbf{begin} \cdots \textbf{end}
\end{array} \right) \right) & \text{Behaviour Part}
\end{array} \right.
\end{array}
$$

[Link Definition]

$$
= \left\{
\begin{array}{l}
\Omega \left(R_{wrt} \left(
\begin{array}{l}
\left(
\begin{array}{l}
\mathbf{process}\ \ P_1 \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_{11} : T_{11} \,;\cdots s_{1m_1} : T_{1m_1} \mid ps_1 \,] \\
\quad Init == [\, (StPar)' \mid pi_1 \,] \\
\quad Pars == [\, decl_1 \mid p_1 \,] \\
\quad Op\_s_{11} == [\, \Xi StPar \,;\, s_{11}! : T_{11} \mid s_{11}! = s_{11} \,] \\
\quad \ldots \\
\quad Op\_s_{1m_1} == [\, \Xi StPar \,;\, s_{1m_1}! : T_{1m_1} \mid s_{1m_1}! = s_{1m_1} \,] \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}
\right) \\
\ldots \\
\left(
\begin{array}{l}
\mathbf{process}\ \ P_n \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_{n1} : T_{n1} \,;\cdots s_{nm_n} : T_{nm_n} \mid ps_n \,] \\
\quad Init == [\, (StPar)' \mid pi_n \,] \\
\quad Pars == [\, decl_n \mid p_n \,] \\
\quad Op\_s_{n1} == [\, \Xi StPar \,;\, s_{n1}! : T_{n1} \mid s_{n1}! = s_{n1} \,] \\
\quad \ldots \\
\quad Op\_s_{nm_n} == [\, \Xi StPar \,;\, s_{nm_n}! : T_{nm_n} \mid s_{nm_n}! = s_{nm_n} \,] \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}
\right)
\end{array}
\right)\right) \\
\Phi \left(R_{wrt} \left(
\begin{array}{l}
\left(
\begin{array}{l}
\mathbf{process}\ \ P_1 \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_{11} : T_{11} \,;\cdots s_{1m_1} : T_{1m_1} \mid ps_1 \,] \\
\quad Init == [\, (StPar)' \mid pi_1 \,] \\
\quad Pars == [\, decl_1 \mid p_1 \,] \\
\quad Op\_s_{11} == [\, \Xi StPar \,;\, s_{11}! : T_{11} \mid s_{11}! = s_{11} \,] \\
\quad \ldots \\
\quad Op\_s_{1m_1} == [\, \Xi StPar \,;\, s_{1m_1}! : T_{1m_1} \mid s_{1m_1}! = s_{1m_1} \,] \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}
\right) \\
\ldots \\
\left(
\begin{array}{l}
\mathbf{process}\ \ P_n \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\, StPar == [\, s_{n1} : T_{n1} \,;\cdots s_{nm_n} : T_{nm_n} \mid ps_n \,] \\
\quad Init == [\, (StPar)' \mid pi_n \,] \\
\quad Pars == [\, decl_n \mid p_n \,] \\
\quad Op\_s_{n1} == [\, \Xi StPar \,;\, s_{n1}! : T_{n1} \mid s_{n1}! = s_{n1} \,] \\
\quad \ldots \\
\quad Op\_s_{nm_n} == [\, \Xi StPar \,;\, s_{nm_n}! : T_{nm_n} \mid s_{nm_n}! = s_{nm_n} \,] \\
\quad \bullet\, A \\
\mathbf{end}
\end{array}
\right)
\end{array}
\right)\right)
\end{array}
\right.
$$

[Additional Schemas R_{wrt} Rule 23]

$$
= \left\{
\begin{array}{ll}
\Omega & \left(
\begin{array}{l}
\left(\begin{array}{l}
\textbf{process}\ P_1 \mathrel{\widehat=} \textbf{begin}\\
\quad \textbf{state}\, P_1\_StPar == [\, P_1\_s_{11} : T_{11} ; \cdots P_1\_s_{1m_1} : T_{1m_1} \mid ps_1 \,]\\
\quad P_1\_Init == [\, (P_1\_StPar)' \mid pi_1 \,]\\
\quad P_1\_Pars == [\, decl_1 \mid p_1 \,]\\
\quad P_1\_Op\_s_{11} == [\, \Xi P_1\_StPar ; P_1\_s_{11}! : T_{11} \mid\\
\qquad P_1\_s_{11}! = P_1\_s_{11} \,]\\
\quad \ldots\\
\quad P_1\_Op\_s_{1m_1} == [\, \Xi P_1\_StPar ; P_1\_s_{1m_1}! : T_{1m_1} \mid\\
\qquad P_1\_s_{1m_1}! = P_1\_s_{1m_1} \,]\\
\quad \bullet\ R_{wrt}(A)\\
\textbf{end}
\end{array}\right)\\
\ldots\\
\left(\begin{array}{l}
\textbf{process}\ P_n \mathrel{\widehat=} \textbf{begin}\\
\quad \textbf{state}\, P_n\_StPar == [\, P_n\_s_{n1} : T_{n1} ; \cdots P_n\_s_{nm_n} : T_{nm_n} \mid\\
\qquad ps_n \,]\\
\quad P_n\_Init == [\, (P_n\_StPar)' \mid pi_n \,]\\
\quad P_n\_Pars == [\, decl_n \mid p_n \,]\\
\quad P_n\_Op\_s_{n1} == [\, \Xi P_n\_StPar ; P_n\_s_{n1}! : T_{n1} \mid\\
\qquad P_n\_s_{n1}! = P_n\_s_{n1} \,]\\
\quad \ldots\\
\quad P_n\_Op\_s_{nm_n} == [\, \Xi P_n\_StPar ; P_n\_s_{nm_n}! : T_{nm_n} \mid\\
\qquad P_n\_s_{nm_n}! = P_n\_s_{nm_n} \,]\\
\quad \bullet\ R_{wrt}(A)\\
\textbf{end}
\end{array}\right)
\end{array}
\right)\\[2ex]
\Phi & \left(
\begin{array}{l}
\left(\begin{array}{l}
\textbf{process}\ P_1 \mathrel{\widehat=} \textbf{begin}\\
\quad \textbf{state}\, P_1\_StPar == [\, P_1\_s_{11} : T_{11} ; \cdots P_1\_s_{1m_1} : T_{1m_1} \mid ps_1 \,]\\
\quad P_1\_Init == [\, (P_1\_StPar)' \mid pi_1 \,]\\
\quad P_1\_Pars == [\, decl_1 \mid p_1 \,]\\
\quad P_1\_Op\_s_{11} == [\, \Xi P_1\_StPar ; P_1\_s_{11}! : T_{11} \mid\\
\qquad P_1\_s_{11}! = P_1\_s_{11} \,]\\
\quad \ldots\\
\quad P_1\_Op\_s_{1m_1} == [\, \Xi P_1\_StPar ; P_1\_s_{1m_1}! : T_{1m_1} \mid\\
\qquad P_1\_s_{1m_1}! = P_1\_s_{1m_1} \,]\\
\quad \bullet\ R_{wrt}(A)\\
\textbf{end}
\end{array}\right)\\
\ldots\\
\left(\begin{array}{l}
\textbf{process}\ P_n \mathrel{\widehat=} \textbf{begin}\\
\quad \textbf{state}\, P_n\_StPar == [\, P_n\_s_{n1} : T_{n1} ; \cdots P_n\_s_{nm_n} : T_{nm_n} \mid\\
\qquad ps_n \,]\\
\quad P_n\_Init == [\, (P_n\_StPar)' \mid pi_n \,]\\
\quad P_n\_Pars == [\, decl_n \mid p_n \,]\\
\quad P_n\_Op\_s_{n1} == [\, \Xi P_n\_StPar ; P_n\_s_{n1}! : T_{n1} \mid\\
\qquad P_n\_s_{n1}! = P_n\_s_{n1} \,]\\
\quad \ldots\\
\quad P_n\_Op\_s_{nm_n} == [\, \Xi P_n\_StPar ; P_n\_s_{nm_n}! : T_{nm_n} \mid\\
\qquad P_n\_s_{nm_n}! = P_n\_s_{nm_n} \,]\\
\quad \bullet\ R_{wrt}(A)\\
\textbf{end}
\end{array}\right)
\end{array}
\right)
\end{array}
\right.
$$

[Renaming R_{wrt} Rule 24]

$$
= \left\{
\begin{array}{l}
\Omega_3 \left(\Omega_2 \left(
\begin{array}{l}
P_1\_StPar == [\, P_1\_s_{11} : T_{11} \,; \cdots P_1\_s_{1m_1} : T_{1m_1} \mid ps_1 \,] \\
\ldots \\
P_n\_StPar == [\, P_n\_s_{n1} : T_{n1} \,; \cdots P_n\_s_{nm_n} : T_{nm_n} \mid ps_n \,] \\
State == P_1\_StPar \land \cdots \land P_n\_StPar \\
Init == [\, (State)' \mid pi_1 \land \cdots \land pi_n \,] \\
P1\_Pars == [P_1\_Pars.decl_1 \,; \Xi P_2\_StPar \,; \ldots ; \Xi P_n\_StPar \mid \\
\qquad P_1\_Pars.p_1] \\
\ldots \\
Pn\_Pars == [P_n\_Pars.decl_n \,; \Xi P_1\_StPar \,; \ldots ; \\
\qquad \Xi P_{n-1}\_StPar \mid P_n\_Pars.p_n] \\
P_1\_Op\_s_{11} == [\, \Xi State \,; P_1\_s_{11}! : T_{11} \mid P_1\_s_{11}! = P_1\_s_{11} \,] \\
\ldots \\
P_1\_Op\_s_{1m_1} == [\, \Xi State \,; P_1\_s_{1m_1}! : T_{1m_1} \mid \\
\qquad P_1\_s_{1m_1}! = P_1\_s_{1m_1} \,] \\
\ldots \\
P_n\_Op\_s_{n1} == [\, \Xi State \,; P_n\_s_{n1}! : T_{n1} \mid P_n\_s_{n1}! = P_n\_s_{n1} \,] \\
\ldots \\
P_n\_Op\_s_{nm_n} == [\, \Xi State \,; P_n\_s_{nm_n}! : T_{nm_n} \mid \\
\qquad P_n\_s_{nm_n}! = P_n\_s_{nm_n} \,]
\end{array}
\right) \right) \\
\left\{
\begin{array}{l}
P_1 = \Phi \left(R_{wrt} \left(P_1.A \right) \right) \\
\ldots \\
P_n = \Phi \left(R_{wrt} \left(P_n.A \right) \right)
\end{array}
\right.
\end{array}
\right.
$$

[Ω_1 Rule 1 and Φ Rule 8]

$$
= \left\{
\begin{array}{l}
\Omega_3 \left(
\begin{array}{l}
P_1\_StPar \mathrel{\widehat=} [\, P_1\_s_{11} : T_{11} \,; \cdots P_1\_s_{1m_1} : T_{1m_1} \mid ps_1 \,] \\
\ldots \\
P_n\_StPar \mathrel{\widehat=} [\, P_n\_s_{n1} : T_{n1} \,; \cdots P_n\_s_{nm_n} : T_{nm_n} \mid ps_n \,] \\
State \mathrel{\widehat=} P_1\_StPar \land \cdots \land P_n\_StPar \\
Init \mathrel{\widehat=} [\, State' \mid pi_1 \land \cdots \land pi_n \,] \\
P1\_Pars \mathrel{\widehat=} [P_1\_Pars.decl_1 \,; \Xi P_2\_StPar \,; \ldots ; \Xi P_n\_StPar \mid \\
\qquad P_1\_Pars.p_1] \\
\ldots \\
Pn\_Pars \mathrel{\widehat=} [P_n\_Pars.decl_n \,; \Xi P_1\_StPar \,; \ldots ; \Xi P_{n-1}\_StPar \mid \\
\qquad P_n\_Pars.p_n] \\
P_1\_Op\_s_{11} \mathrel{\widehat=} [\, \Xi State \,; P_1\_s_{11}! : T_{11} \mid P_1\_s_{11}! = P_1\_s_{11} \,] \\
\ldots \\
P_1\_Op\_s_{1m_1} \mathrel{\widehat=} [\, \Xi State \,; P_1\_s_{1m_1}! : T_{1m_1} \mid P_1\_s_{1m_1}! = P_1\_s_{1m_1} \,] \\
\ldots \\
P_n\_Op\_s_{n1} \mathrel{\widehat=} [\, \Xi State \,; P_n\_s_{n1}! : T_{n1} \mid P_n\_s_{n1}! = P_n\_s_{n1} \,] \\
\ldots \\
P_n\_Op\_s_{nm_n} \mathrel{\widehat=} [\, \Xi State \,; P_n\_s_{nm_n}! : T_{nm_n} \mid P_n\_s_{nm_n}! = P_n\_s_{nm_n} \,]
\end{array}
\right) \\
\left\{
\begin{array}{l}
P_1 = \Phi \left(R_{wrt} \left(P_1.A \right) \right) \\
\ldots \\
P_n = \Phi \left(R_{wrt} \left(P_n.A \right) \right)
\end{array}
\right.
\end{array}
\right.
$$

[Ω_2 Rule 4 and Ω_2 Rule 5]

$\Box$

E.5 Compound Processes

The state part of explicitly defined processes is linked to Z by Link Rule 8 and 9. Therefore, when linking compound processes, the link of the state part is omitted and only the behavioural part is linked in this section.

E.5.1 Sequential Composition

Link Rule 10 (Sequential Composition).

$$\begin{aligned}
&\Upsilon(P\,;Q)\\
&=\begin{cases}\Omega(R_{wrt}(P\,;Q)) & \text{State Part}\\[2ex] \Phi(R_{wrt}(P\,;Q)) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&=\begin{cases}\Omega(R_{wrt}(P)\,;R_{wrt}(Q))\\[2ex] \Phi(R_{wrt}(P)\,;R_{wrt}(Q))\end{cases} && [R_{wrt}\ \text{Rule 22}]\\
&=\begin{cases}\Omega_3(\Omega_2(\Omega_1(R_{wrt}(P)\,;R_{wrt}(Q))))\\[2ex] \Phi(R_{wrt}(P))\,;\Phi(R_{wrt}(Q))\end{cases} && [\Omega\ \text{Definition and}\ \Phi\ \text{Rule 9}]\\
&=\Phi(R_{wrt}(P))\,;\Phi(R_{wrt}(Q)) && \text{[The link of the state part by}\ \Omega\ \text{is omitted.]}\\
&=\Phi(P)\,;\Phi(Q) && [R_{wrt}\ \text{Rule 18}]\\
&=P\,;Q && [\Phi\ \text{Rule 15}]
\end{aligned}$$

□

E.5.2 External Choice

Link Rule 11 (External Choice).

$$\begin{aligned}
&\Upsilon(PA_1 \Box PA_2)\\
&=\begin{cases}\Omega(R_{wrt}(PA_1 \Box PA_2)) & \text{State Part}\\[2ex] \Phi(R_{wrt}(PA_1 \Box PA_2)) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&=\begin{cases}\Omega(R_{wrt}(PA_1) \Box R_{wrt}(PA_2))\\[2ex] \Phi(R_{wrt}(PA_1) \Box R_{wrt}(PA_2))\end{cases} && [R_{wrt}\ \text{Rule 22}]\\
&=\begin{cases}\Omega_3(\Omega_2(\Omega_1(R_{wrt}(PA_1) \Box R_{wrt}(PA_2))))\\[2ex] \Phi(R_{wrt}(PA_1)) \Box \Phi(R_{wrt}(PA_2))\end{cases} && [\Omega\ \text{Definition and}\ \Phi\ \text{Rule 9}]\\
&=\Phi(PA_1) \Box \Phi(PA_2) && [R_{wrt}\ \text{Rule 18}]\\
&=PA_1 \Box PA_2 && [\Phi\ \text{Rule 15}]
\end{aligned}$$

□

E.5.3 Internal Choice

Link Rule 12 (Internal Choice).

$$\Upsilon(P \sqcap Q)$$

$$= \begin{cases} \Omega\,(R_{wrt}\,(P \sqcap Q)) & \text{State Part} \\ \\ \Phi\,(R_{wrt}\,(P \sqcap Q)) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\,(R_{wrt}\,(P) \sqcap R_{wrt}\,(Q)) \\ \\ \Phi\,(R_{wrt}\,(P) \sqcap R_{wrt}\,(Q)) \end{cases} \qquad [R_{wrt} \text{ Rule 22}]$$

$$= \begin{cases} \Omega_3\,(\Omega_2\,(\Omega_1\,(R_{wrt}\,(P) \sqcap R_{wrt}\,(Q)))) \\ \\ \Phi\,(R_{wrt}\,(P)) \sqcap \Phi\,(R_{wrt}\,(Q)) \end{cases} \qquad [\Omega \text{ Definition and } \Phi \text{ Rule 9}]$$

$$= \Phi\,(P) \sqcap \Phi\,(Q) \qquad [R_{wrt} \text{ Rule 18}]$$

$$= P \sqcap Q \qquad [\Phi \text{ Rule 15}]$$

□

E.5.4 Parallel Composition

Link Rule 13 (Parallel Composition).

$$\Upsilon\,(P \llbracket\, cs \,\rrbracket\, Q)$$

$$= \begin{cases} \Omega\,(R_{wrt}\,(P \llbracket\, cs \,\rrbracket\, Q)) & \text{State Part} \\ \\ \Phi\,(R_{wrt}\,(P \llbracket\, cs \,\rrbracket\, Q)) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\,(R_{wrt}\,(P) \llbracket\, cs \,\rrbracket\, R_{wrt}\,(Q)) \\ \\ \Phi\,(R_{wrt}\,(P) \llbracket\, cs \,\rrbracket\, R_{wrt}\,(Q)) \end{cases} \qquad [R_{wrt} \text{ Rule 22}]$$

$$= \begin{cases} \Omega_3\,(\Omega_2\,(\Omega_1\,(R_{wrt}\,(P) \llbracket\, cs \,\rrbracket\, R_{wrt}\,(Q)))) \\ \\ \Phi\,(R_{wrt}\,(P)) \underset{\Phi(cs)}{\parallel} \Phi\,(R_{wrt}\,(Q)) \end{cases} \qquad [\Omega \text{ Definition and } \Phi \text{ Rule 12}]$$

$$= \Phi\,(P) \underset{\Phi(cs)}{\parallel} \Phi\,(Q) \qquad [R_{wrt} \text{ Rule 18}]$$

$$= P \underset{\Phi(cs)}{\parallel} Q \qquad [\Phi \text{ Rule 15}]$$

□

E.5.5 Interleaving

Link Rule 14 (Interleaving).

$$\Upsilon\,(P \;|||\; Q)$$

$$= \begin{cases} \Omega\,(R_{wrt}\,(P \;|||\; Q)) & \text{State Part} \\ \\ \Phi\,(R_{wrt}\,(P \;|||\; Q)) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\,(R_{wrt}\,(P) \;|||\; R_{wrt}\,(Q)) \\ \\ \Phi\,(R_{wrt}\,(P) \;|||\; R_{wrt}\,(Q)) \end{cases} \qquad [R_{wrt} \text{ Rule 22}]$$

$$= \begin{cases} \Omega_3\,(\Omega_2\,(\Omega_1\,(R_{wrt}\,(P) \;|||\; R_{wrt}\,(Q)))) \\ \\ \Phi\,(R_{wrt}\,(P)) \;|||\; \Phi\,(R_{wrt}\,(Q)) \end{cases} \qquad [\Omega \text{ Definition and } \Phi \text{ Rule 13}]$$

$$
\begin{aligned}
&= \Phi(P) \mathbin{|||} \Phi(Q) && [R_{wrt} \text{ Rule 18}]\\
&= P \mathbin{|||} Q && [\Phi \text{ Rule 15}]
\end{aligned}
$$

□

E.5.6 Hiding

Link Rule 15 (Hiding).

$$
\begin{aligned}
&\Upsilon(P \setminus cs)\\
&= \begin{cases} \Omega(R_{wrt}(P \setminus cs)) & \text{State Part}\\[1ex] \Phi(R_{wrt}(P \setminus cs)) & \text{Behaviour Part} \end{cases} && [\text{Link Definition}]\\
&= \begin{cases} \Omega(R_{wrt}(P) \setminus cs)\\[1ex] \Phi(R_{wrt}(P) \setminus cs) \end{cases} && [R_{wrt} \text{ Rule 22}]\\
&= \Phi(R_{wrt}(P)) \setminus \Phi(cs) && [\Phi \text{ Rule 14}]\\
&= \Phi(P) \setminus \Phi(cs) && [R_{wrt} \text{ Rule 18}]\\
&= P \setminus \Phi(cs) && [\Phi \text{ Rule 15}]
\end{aligned}
$$

□

E.5.7 Unnamed Parametrised Process Invocation

Link Rule 16 (Unnamed Parametrised Process Invocation). An unnamed parametrised process invocation is rewritten to a named parametrised process and its invocation. Then they are linked by the rules of the parametrised process and its invocation.

$$
\begin{aligned}
&\Upsilon((x : T \bullet P)(e))\\
&= \begin{cases} \Omega(R_{wrt}((x : T \bullet P)(e))) & \text{State Part}\\[1ex] \Phi(R_{wrt}((x : T \bullet P)(e))) & \text{Behaviour Part} \end{cases} && [\text{Link Definition}]\\
&= \begin{cases} \Omega\begin{pmatrix} R_{wrt}(\textbf{process } UPP \mathrel{\widehat=} x : T \bullet P)\\ R_{wrt}(UPP(e)) \end{pmatrix} & \text{State Part}\\[2ex] \Phi\begin{pmatrix} R_{wrt}(\textbf{process } UPP \mathrel{\widehat=} x : T \bullet P)\\ R_{wrt}(UPP(e)) \end{pmatrix} & \text{Behaviour Part} \end{cases} && [R_{wrt} \text{ Rule 13}]
\end{aligned}
$$

Then $R_{wrt}(\textbf{process } UPP \mathrel{\widehat=} x : T \bullet P)$ is linked by Link Rule 29 and $R_{wrt}(UPP(e))$ is linked by Link Rule 17.

It is worth noting that this link rule finally results in a set of new explicitly defined processes and they should be linked to the state part as well by Link Rule 9 in the later stage. □

E.5.8 Parametrised Process Invocation

Link Rule 17 (Parametrised Process Invocation). For the parametrised process invocation $PP(ax)$ (where ax is a variable name or an expression) in *Circus*, it is linked below.

$$\begin{aligned}
&\Upsilon\left(PP(ax)\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(PP(ax)\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left(PP(ax)\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \begin{cases} \Omega\left(\begin{array}{ll} & (ax == x_1)\ \&\ PP\_x_1\\ \Box & (ax == x_2)\ \& PP\_x_2\\ & \ldots\\ \Box & (ax == x_n)\ \& PP\_x_n\end{array}\right)\\ \\ \Phi\left(\begin{array}{ll} & (ax == x_1)\ \&\ PP\_x_1\\ \Box & (ax == x_2)\ \& PP\_x_2\\ & \ldots\\ \Box & (ax == x_n)\ \& PP\_x_n\end{array}\right)\end{cases} && [R_{wrt}\ \text{Rule 12}]\\
&= \Phi\left(\begin{array}{ll} & (ax == x_1)\ \&\ PP\_x_1\\ \Box & (ax == x_2)\ \& PP\_x_2\\ & \ldots\\ \Box & (ax == x_n)\ \& PP\_x_n\end{array}\right) && \text{[No state part]}
\end{aligned}$$

Thus, for a specific value of ax (assume $ax = x_i$), the invocation $PP(ax)$ becomes

$$\begin{aligned}
&\Upsilon\left(PP(x_i)\right)\\
&= \Phi\left(\begin{array}{ll} & (ax == x_1)\ \&\ PP\_x_1\\ \Box & (ax == x_2)\ \& PP\_x_2\\ & \ldots\\ \Box & (ax == x_n)\ \& PP\_x_n\end{array}\right)\\
&= \Phi\left(PP\_x_i\right) && \text{[External choice elimination Lemma C.3.1]}\\
&= \Phi\left(P[x_i/x]\right) && [P\_x_i\ \text{definition in}\ R_{wrt}\ \text{Rule 11}]
\end{aligned}$$

Finally, the invocation of $PP(x_i)$ is the invocation of an explicitly defined process that is got by substituting x_i for x in P. □

E.5.9 Process Invocation

Link Rule 18 (Process Invocation).

$$\begin{aligned}
&\Upsilon(P)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(P\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left(P\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \begin{cases} \Omega\left(P\right)\\ \\ \Phi\left(P\right)\end{cases} && [\text{Process invocation}\ R_{wrt}\ \text{Rule 22}]\\
&= \Phi\left(P\right) && \text{[No state part]}\\
&= P && [\Phi\ \text{Rule 15}]
\end{aligned}$$

□

E.5.10 Unnamed Indexed Process Invocation

Link Rule 19 (Unnamed Indexed Process Invocation). An unnamed indexed process invocation is rewritten to a named indexed process and its invocation. Then they are linked by the rules of the indexed process and its invocation.

$$
\begin{aligned}
&\Upsilon\left((x : T \odot P)\lfloor e \rfloor\right) \\
&= \begin{cases} \Omega\left(R_{wrt}\left((x : T \odot P)\lfloor e \rfloor\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left((x : T \odot P)\lfloor e \rfloor\right)\right) & \text{Behaviour Part} \end{cases} && [\text{Link Definition}] \\
&= \begin{cases} \Omega\begin{pmatrix} R_{wrt}\left(\mathbf{process}\ \ UIP \mathrel{\widehat=} x : T \odot P\right) \\ R_{wrt}\left(UIP\lfloor e \rfloor\right) \end{pmatrix} & \text{State Part} \\ \\ \Phi\begin{pmatrix} R_{wrt}\left(\mathbf{process}\ \ UIP \mathrel{\widehat=} x : T \odot P\right) \\ R_{wrt}\left(UIP\lfloor e \rfloor\right) \end{pmatrix} & \text{Behaviour Part} \end{cases} && [R_{wrt}\ \text{Rule 16}]
\end{aligned}
$$

Then $R_{wrt}\left(\mathbf{process}\ \ UIP \mathrel{\widehat=} x : T \odot P\right)$ is linked by Link Rule 30 and $R_{wrt}\left(UIP\lfloor e \rfloor\right)$ is linked by Link Rule 20. □

E.5.11 Indexed Process Invocation

Link Rule 20 (Indexed Process Invocation). For the indexed process invocation $IP\lfloor ax \rfloor$ (where ax is a variable name or an expression) in *Circus*, it is linked below.

$$
\begin{aligned}
&\Upsilon\left(IP\lfloor ax \rfloor\right) \\
&= \begin{cases} \Omega\left(R_{wrt}\left(IP\lfloor ax \rfloor\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left(IP\lfloor ax \rfloor\right)\right) & \text{Behaviour Part} \end{cases} && [\text{Link Definition}] \\
&= \begin{cases} \Omega\begin{pmatrix} & (ax == x_1)\ \&\ IP\_x_1 \\ \Box & (ax == x_2)\ \& IP\_x_2 \\ & \cdots \\ \Box & (ax == x_n)\ \& IP\_x_n \end{pmatrix} \\ \\ \Phi\begin{pmatrix} & (ax == x_1)\ \&\ IP\_x_1 \\ \Box & (ax == x_2)\ \& IP\_x_2 \\ & \cdots \\ \Box & (ax == x_n)\ \& IP\_x_n \end{pmatrix} \end{cases} && [R_{wrt}\ \text{Rule 15}] \\
&= \Phi\begin{pmatrix} & (ax == x_1)\ \&\ IP\_x_1 \\ \Box & (ax == x_2)\ \& IP\_x_2 \\ & \cdots \\ \Box & (ax == x_n)\ \& IP\_x_n \end{pmatrix} && [\text{No state part}]
\end{aligned}
$$

Thus, for a specific value of ax (assume $ax = x_i$), the invocation $IP\lfloor ax \rfloor$ becomes

$$
\begin{aligned}
&\Upsilon\left(IP\lfloor ax \rfloor\right) \\
&= \Phi\begin{pmatrix} & (ax == x_1)\ \&\ IP\_x_1 \\ \Box & (ax == x_2)\ \& IP\_x_2 \\ & \cdots \\ \Box & (ax == x_n)\ \& IP\_x_n \end{pmatrix} && [\text{No state part}] \\
&= \Phi\left(IP\_x_i\right) && [\text{External choice elimination Lemma C.3.1}] \\
&= \Phi\left(P[x_i/x]\right) && [P\_x_i\ \text{definition in}\ R_{wrt}\ \text{Rule 14}]
\end{aligned}
$$

Finally, the invocation of $IP\lfloor x_i \rfloor$ is the invocation of an explicitly defined process that is got by substituting x_i for x in IP. □

E.5.12 Renaming Operator

Link Rule 21 (Renaming Operator). Provided P is a reference to an explicitly defined process or an indexed process, then

$$\begin{array}{l}
\Upsilon\,(\mathbf{process}\,RP \mathrel{\widehat=} P[c_{old} := c_{new}]) \\
= \left\{ \begin{array}{ll} \Omega\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} P[c_{old} := c_{new}])) & \text{State Part} \\ & \\ \Phi\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} P[c_{old} := c_{new}])) & \text{Behaviour Part} \end{array} \right. \\
\hfill [\text{Link Definition}] \\
= \left\{ \begin{array}{ll} \Omega\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} F_{Ren}\,(B(P), \{(c_{old}, c_{new})\}))) & \text{State Part} \\ & \\ \Phi\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} F_{Ren}\,(B(P), \{(c_{old}, c_{new})\}))) & \text{Behaviour Part} \end{array} \right. \\
\hfill [R_{wrt}\ \text{Rule 19}]
\end{array}$$

By this rule, the original RP defined as a renamed process has become an explicitly defined process (if P is an explicitly process) or a set of explicitly defined processes (if P is an indexed process). And therefore it can be linked further by Link Rule 9. □

Link Rule 22 (Explicitly Defined Processes with Renaming). Provided P is a reference to an explicitly defined process, then

$$\begin{array}{l}
\Upsilon\,(\mathbf{process}\,RP \mathrel{\widehat=} P[c_{old} := c_{new}]) \\
= \left\{ \begin{array}{ll} \Omega\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} F_{Ren}\,(B(P), \{(c_{old}, c_{new})\}))) & \text{State Part} \\ & \\ \Phi\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} F_{Ren}\,(B(P), \{(c_{old}, c_{new})\}))) & \text{Behaviour Part} \end{array} \right. \\
\hfill [\text{Link Rule 21}] \\
= \left\{ \begin{array}{ll} \Omega\,(\mathbf{process}\,RP \mathrel{\widehat=} R_{wrt}\,(F_{Ren}\,(B(P), \{(c_{old}, c_{new})\}))) & \text{State Part} \\ & \\ \Phi\,(\mathbf{process}\,RP \mathrel{\widehat=} R_{wrt}\,(F_{Ren}\,(B(P), \{(c_{old}, c_{new})\}))) & \text{Behaviour Part} \end{array} \right. \\
\hfill [\text{Link Rule 20}]
\end{array}$$

Then it can be linked further by Link Rule 9. □

Link Rule 23 (Indexed Processes with Renaming). Provided IP is a reference to an index process, then

$$\begin{array}{l}
\Upsilon\,(\mathbf{process}\,RP \mathrel{\widehat=} IP[c\_i := d]) \\
= \left\{ \begin{array}{ll} \Omega\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} F_{Ren}\,(B(IP), \{(c\_i, d)\}))) & \text{State Part} \\ & \\ \Phi\,(R_{wrt}\,(\mathbf{process}\,RP \mathrel{\widehat=} F_{Ren}\,(B(IP), \{(c\_i, d)\}))) & \text{Behaviour Part} \end{array} \right. \\
\hfill [\text{Link Rule 21}]
\end{array}$$

$$= \begin{cases} \Omega \left(\begin{array}{l} \textbf{process } RP\_i_1 \mathrel{\widehat=} R_{wrt}\left(F_{Ren}\left(B(P), \{(c, d.i_1)\}\right)\right) \\ \vdots \\ \textbf{process } RP\_i_n \mathrel{\widehat=} R_{wrt}\left(F_{Ren}\left(B(P), \{(c, d.i_n)\}\right)\right) \end{array} \right) & \text{State Part} \\ \Phi \left(\begin{array}{l} \textbf{process } RP\_i_1 \mathrel{\widehat=} R_{wrt}\left(F_{Ren}\left(B(P), \{(c, d.i_1)\}\right)\right) \\ \vdots \\ \textbf{process } RP\_i_n \mathrel{\widehat=} R_{wrt}\left(F_{Ren}\left(B(P), \{(c, d.i_n)\}\right)\right) \end{array} \right) & \text{Behaviour Part} \end{cases}$$

$[R_{wrt}$ Rule 21]

Then these explicitly defined processes can be linked further by Link Rule 9. □

E.5.13 Iterated Sequential Composition

Link Rule 24 (Iterated Sequential Composition).

$$\Upsilon\left(;\, x : T \bullet P(x)\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(;\, x : T \bullet P(x)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(;\, x : T \bullet P(x)\right)\right) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(;\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \\ \Phi\left(;\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \end{cases} \qquad \text{[Iterated sequential composition } R_{wrt} \text{ Rule 22]}$$

$$= \Phi\left(;\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \qquad \text{[Only behavioural part in this construct]}$$

$$= ;_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) \qquad [\Phi \text{ Rule 16}]$$

□

E.5.14 Iterated External Choice

Link Rule 25 (Iterated External Choice).

$$\Upsilon\left(\Box\, x : T \bullet P(x)\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\Box\, x : T \bullet P(x)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\Box\, x : T \bullet P(x)\right)\right) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(\Box\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \\ \Phi\left(\Box\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \end{cases} \qquad \text{[Iterated external choice } R_{wrt} \text{ Rule 22]}$$

$$= \Phi\left(\Box\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \qquad \text{[Only behavioural part in this construct]}$$

$$= \Box_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) \qquad [\Phi \text{ Rule 17}]$$

□

E.5.15 Iterated Internal Choice

Link Rule 26 (Iterated Internal Choice).

$$\Upsilon\left(\sqcap\, x : T \bullet P(x)\right)$$

$$
\begin{aligned}
&= \begin{cases} \Omega\left(R_{wrt}\left(\sqcap x : T \bullet P(x)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\sqcap x : T \bullet P(x)\right)\right) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \begin{cases} \Omega\left(\sqcap x : T \bullet R_{wrt}\left(P(x)\right)\right) \\ \Phi\left(\sqcap x : T \bullet R_{wrt}\left(P(x)\right)\right) \end{cases} && \text{[Iterated internal choice } R_{wrt} \text{ Rule 22]} \\
&= \Phi\left(\sqcap x : T \bullet R_{wrt}\left(P(x)\right)\right) && \text{[Only behavioural part in this construct]} \\
&= \sqcap_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) && \text{[}\Phi\text{ Rule 18]}
\end{aligned}
$$

□

E.5.16 Iterated Parallel Composition

Link Rule 27 (Iterated Parallel Composition).

$$
\begin{aligned}
&\Upsilon\left(\llbracket CS \rrbracket\, x : T \bullet P(x)\right) \\
&= \begin{cases} \Omega\left(R_{wrt}\left(\llbracket CS \rrbracket\, x : T \bullet P(x)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\llbracket CS \rrbracket\, x : T \bullet P(x)\right)\right) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \begin{cases} \Omega\left(\llbracket CS \rrbracket\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \\ \Phi\left(\llbracket CS \rrbracket\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \end{cases} \\
& && \text{[Iterated parallel composition } R_{wrt} \text{ Rule 22]} \\
&= \Phi\left(\llbracket CS \rrbracket\, x : T \bullet R_{wrt}\left(P(x)\right)\right) && \text{[Only behavioural part in this construct]} \\
&= \underset{CS}{\|}_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) && \text{[}\Phi\text{ Rule 19]}
\end{aligned}
$$

□

E.5.17 Iterated Interleaving

Link Rule 28 (Iterated Interleaving).

$$
\begin{aligned}
&\Upsilon\left(|||\, x : T \bullet P(x)\right) \\
&= \begin{cases} \Omega\left(R_{wrt}\left(|||\, x : T \bullet P(x)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(|||\, x : T \bullet P(x)\right)\right) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \begin{cases} \Omega\left(|||\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \\ \Phi\left(|||\, x : T \bullet R_{wrt}\left(P(x)\right)\right) \end{cases} && \text{[Iterated parallel composition } R_{wrt} \text{ Rule 22]} \\
&= \Phi\left(|||\, x : T \bullet R_{wrt}\left(P(x)\right)\right) && \text{[Only behavioural part in this construct]} \\
&= |||_{x:\Phi(T)} \bullet \Phi\left(R_{wrt}\left(P(x)\right)\right) && \text{[}\Phi\text{ Rule 19]}
\end{aligned}
$$

□

E.6 Parametrised Processes

Link Rule 29 (Parametrised Processes).

$$
\begin{aligned}
&\Upsilon\,(\textbf{process}\ \ PP \mathrel{\widehat{=}} x : T \bullet P) \\
&= \begin{cases} \Omega\,(R_{wrt}\,(\textbf{process}\ \ PP \mathrel{\widehat{=}} x : T \bullet P)) & \text{State Part} \\ \\ \Phi\,(R_{wrt}\,(\textbf{process}\ \ PP \mathrel{\widehat{=}} x : T \bullet P)) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \begin{cases} \Omega \begin{pmatrix} R_{wrt}\,(\textbf{process}\ \ PP\_x_1 \mathrel{\widehat{=}} P[x_1/x]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ \ PP\_x_n \mathrel{\widehat{=}} P[x_n/x]) \end{pmatrix} \\ \\ \Phi \begin{pmatrix} R_{wrt}\,(\textbf{process}\ \ PP\_x_1 \mathrel{\widehat{=}} P[x_1/x]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ \ PP\_x_n \mathrel{\widehat{=}} P[x_n/x]) \end{pmatrix} \end{cases} \\
& && \text{[Parametrised process } R_{wrt} \text{ Rule 11]}
\end{aligned}
$$

The parametrised process *PP* is rewritten to a set of explicitly defined processes that are linked to the *CSP* ∥ *B* program by Link Rule 9. □

E.7 Indexed Processes

Link Rule 30 (Indexed Processes).

$$
\begin{aligned}
&\Upsilon\,(\textbf{process}\ \ IP \mathrel{\widehat{=}} i : T \odot P) \\
&= \begin{cases} \Omega\,(R_{wrt}\,(\textbf{process}\ \ IP \mathrel{\widehat{=}} i : T \odot P)) & \text{State Part} \\ \\ \Phi\,(R_{wrt}\,(\textbf{process}\ \ IP \mathrel{\widehat{=}} i : T \odot P)) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \begin{cases} \Omega \begin{pmatrix} R_{wrt}\,(\textbf{process}\ \ IP\_i_1 \mathrel{\widehat{=}} P[c := c\_i.i_1]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ \ IP\_i_n \mathrel{\widehat{=}} P[c := c\_i.i_n]) \end{pmatrix} \\ \\ \Phi \begin{pmatrix} R_{wrt}\,(\textbf{process}\ \ IP\_i_1 \mathrel{\widehat{=}} P[c := c\_i.i_1]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ \ IP\_i_n \mathrel{\widehat{=}} P[c := c\_i.i_n]) \end{pmatrix} \end{cases} \\
& && \text{[Indexed process } R_{wrt} \text{ Rule 14]} \\
&= \begin{cases} \Omega \begin{pmatrix} R_{wrt}\,(\textbf{process}\ \ IP\_i_1 \mathrel{\widehat{=}} F_{Ren}(P, \{(c, c\_i.i_1)\})) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ \ IP\_i_n \mathrel{\widehat{=}} F_{Ren}(P, \{(c, c\_i.i_n)\})) \end{pmatrix} \\ \\ \Phi \begin{pmatrix} R_{wrt}\,(\textbf{process}\ \ IP\_i_1 \mathrel{\widehat{=}} F_{Ren}(P, \{(c, c\_i.i_1)\})) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ \ IP\_i_n \mathrel{\widehat{=}} F_{Ren}(P, \{(c, c\_i.i_n)\})) \end{pmatrix} \end{cases} \\
& && \text{[Renaming operator } R_{wrt} \text{ Rule 19]}
\end{aligned}
$$

The indexed process *IP* is rewritten to a set of explicitly defined processes that are linked to the *CSP* ∥ *B* program by Link Rule 9. □

E.8 Actions

E.8.1 Schema Expression as Action

Link Rule 31 (Schema Expression as Action).

$$
\begin{aligned}
&\Upsilon\left(\left(SExp\right)\right)\\
&= \left\{\begin{array}{ll}
\Omega\,(R_{wrt}\,(SExp == [decl\,;\,ins?:T_i\,;\,outs!:T_o \mid p])) & \text{State Part}\\[2ex]
\Phi\left(R_{wrt}\left(\left(SExp\right)\right)\right) & \text{Behaviour Part}
\end{array}\right.\\
&\hfill[\text{Link Definition and } SExp \text{ schema definition }]\\
&= \left\{\begin{array}{l}
\Omega\,(P\_SExp == [decl\,;\,ins?:T_i\,;\,outs!:T_o \mid p])\\[2ex]
\Phi\left(\left(P\_SExp\right)\right)
\end{array}\right.\\
&\hfill[\text{Renaming } R_{wrt} \text{ Rule 24 and Schema expression } R_{wrt} \text{ Rule 25}]\\
&= \left\{\begin{array}{l}
\Omega_3\left(\Omega_2\left(\begin{array}{l}
P\_SExp == [\,decl\,;\Xi Q_1\_StPar\,;\cdots\Xi Q_n\_StPar;\\
\qquad ins?:T_i\,;\,outs!:T_o \mid p\,]\\
P\_SExp\_fOp == [\Xi P\_StPar\,;\Xi Q_1\_StPar\,;\cdots\Xi Q_n\_StPar;\\
\qquad ins?:T_i \mid \neg\,\mathbf{pre}\,P\_SExp]
\end{array}\right)\right)\\[2ex]
\left(\begin{array}{l}
channel\ P\_SExp : \Phi(T_i).\Phi(T_o)\\
channel\ P\_SExp\_fOp : \Phi(T_i)\\
HIDE\_CSPB = \{\!|P\_SExp, P\_SExp\_fOp|\!\}\\
P\_SExp!ins?outs \rightarrow SKIP \,\Box\, P\_SExp\_fOp!ins \rightarrow \mathbf{div}
\end{array}\right)
\end{array}\right.\\
&\hfill[\Omega_1 \text{ Rule 1 and } \Phi \text{ Rule 21}]\\
&= \left\{\begin{array}{l}
\Omega_3\left(\begin{array}{l}
P\_SExp \mathrel{\widehat=} [\,decl\,;\Xi Q_1\_StPar\,;\cdots\Xi Q_n\_StPar;\\
\qquad ins?:T_i\,;\,outs!:T_o \mid p\,]\\
P\_SExp\_fOp \mathrel{\widehat=} [\Xi P\_StPar\,;\Xi Q_1\_StPar\,;\cdots\Xi Q_n\_StPar;\\
\qquad ins?:T_i \mid \neg\,\mathbf{pre}\,P\_SExp]
\end{array}\right)\\[2ex]
\left(\begin{array}{l}
channel\ P\_SExp : \Phi(T_i).\Phi(T_o)\\
channel\ P\_SExp\_fOp : \Phi(T_i)\\
HIDE\_CSPB = \{\!|P\_SExp, P\_SExp\_fOp|\!\}\\
P\_SExp!ins?outs \rightarrow SKIP \,\Box\, P\_SExp\_fOp!ins \rightarrow \mathbf{div}
\end{array}\right)
\end{array}\right.\\
&\hfill[\Omega_2 \text{ Rule 4}]
\end{aligned}
$$

□

E.8.2 CSP Actions

E.8.2.1 Basic Actions

Link Rule 32 (Basic Actions).

$$
\begin{aligned}
&\Upsilon\,(\mathbf{Skip})\\
&= \left\{\begin{array}{ll}
\Omega\,(R_{wrt}\,(\mathbf{Skip})) & \text{State Part}\\[2ex]
\Phi\,(R_{wrt}\,(\mathbf{Skip})) & \text{Behaviour Part}
\end{array}\right. && [\text{Link Definition}]\\
&= \Phi\,(R_{wrt}\,(\mathbf{Skip})) && [\text{Only behavioural part in this construct}]
\end{aligned}
$$

$$\begin{aligned}
&= \Phi\left(\mathbf{Skip}\right) && [R_{wrt} \text{ Rule } 26]\\
&= \mathit{SKIP} && [\Phi \text{ Rule } 23]
\end{aligned}$$

$$\begin{aligned}
&\Upsilon\left(\mathbf{Stop}\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(\mathbf{Stop}\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left(\mathbf{Stop}\right)\right) & \text{Behaviour Part}\end{cases} && [\text{Link Definition}]\\
&= \Phi\left(R_{wrt}\left(\mathbf{Stop}\right)\right) && [\text{Only behavioural part in this construct}]\\
&= \Phi\left(\mathbf{Stop}\right) && [R_{wrt} \text{ Rule } 26]\\
&= \mathit{STOP} && [\Phi \text{ Rule } 23]
\end{aligned}$$

$$\begin{aligned}
&\Upsilon\left(\mathbf{Chaos}\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(\mathbf{Chaos}\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left(\mathbf{Chaos}\right)\right) & \text{Behaviour Part}\end{cases} && [\text{Link Definition}]\\
&= \Phi\left(R_{wrt}\left(\mathbf{Chaos}\right)\right) && [\text{Only behavioural part in this construct}]\\
&= \Phi\left(\mathbf{Chaos}\right) && [R_{wrt} \text{ Rule } 26]\\
&= \mathbf{div} && [\Phi \text{ Rule } 23]
\end{aligned}$$

□

E.8.2.2 Prefixing

Synchronisation Channel

Link Rule 33 (Synchronisation Channel).

$$\begin{aligned}
&\Upsilon\left(c \to A\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(c \to A\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left(c \to A\right)\right) & \text{Behaviour Part}\end{cases} && [\text{Link Definition}]\\
&= \Phi\left(R_{wrt}\left(c \to A\right)\right) && [\text{Only behavioural part in this construct}]\\
&= \Phi\left(c \to R_{wrt}\left(A\right)\right) && [R_{wrt} \text{ Rule } 27]\\
&= c \to \Phi\left(R_{wrt}\left(A\right)\right) && [\Phi \text{ Rule } 24]
\end{aligned}$$

□

Output Channel

Link Rule 34 (Output Channel). Similarly, for the prefixing $c.e \to A$ or $c!e \to A$ in *Circus*, provided e does not evaluate state variables, it is linked to $c.e \to \Phi\left(R_{wrt}\left(A\right)\right)$ or $c!e \to \Phi\left(R_{wrt}\left(A\right)\right)$. □

Link Rule 35 (Output Channel). However for the prefixing $c.e(s_i, \cdots, s_j) \to A$, if e evaluates state variables such as $s_i, \cdots, s_j$, it is linked below.

$$\Upsilon\left(c.e(s_i, \cdots, s_j) \to A\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(c.e(s_i,\cdots,s_j)\rightarrow A\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left(c.e(s_i,\cdots,s_j)\rightarrow A\right)\right) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \Phi\left(\left(P\_Op\_s_i\right)\rightarrow\cdots\rightarrow\left(P\_Op\_s_j\right)\rightarrow c.e(s_i,\cdots,s_j)\rightarrow R_{wrt}\left(A\right)\right)$$

$$\text{[Only behavioural part, } R_{wrt} \text{ Rule 24 and } R_{wrt} \text{ Rule 27]}$$

$$= \Phi\left(\left(P\_Op\_s_i\right)\right)\rightarrow\ldots\rightarrow\Phi\left(\left(P\_Op\_s_j\right)\right)\rightarrow c.\Phi\left(e(s_i,\cdots,s_j)\right)\rightarrow\Phi\left(R_{wrt}\left(A\right)\right)$$

$$[\Phi \text{ Rule 24]}$$

$$= \begin{cases} \mathit{channel}\ P\_Op\_s_i : \Phi(T_{s_i}) \\ \cdots \\ \mathit{channel}\ P\_Op\_s_j : \Phi(T_{s_j}) \\ HIDE\_CSPB = \{\!|P\_Op\_s_i,\cdots,P\_Op\_s_j|\!\} \\ P\_Op\_s_i?s_i\rightarrow\cdots\rightarrow P\_Op\_s_j?s_j\rightarrow c.\Phi\left(e(s_i,\cdots,s_j)\right)\rightarrow\Phi\left(R_{wrt}\left(A\right)\right) \end{cases}$$

$$[\Phi \text{ Rule 22]}$$

□

Input Channel

Link Rule 36 (Input Channel). A restricted input prefixing $c?x : P \rightarrow A(x)$ in *Circus* is linked below.

$$\Upsilon\left(c?x : P\rightarrow A(x)\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(c?x : P\rightarrow A(x)\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left(c?x : P\rightarrow A(x)\right)\right) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \Phi\left(R_{wrt}\left(c?x : P\rightarrow A(x)\right)\right) \qquad \text{[Only behavioural part in this construct]}$$

$$= \Phi\left(c?x : P\rightarrow R_{wrt}\left(A(x)\right)\right) \qquad \text{[Input Prefix } R_{wrt} \text{ Rule 27]}$$

$$= c?x : \{y \mid y\text{<-}\Phi(T_c), \Phi(P)\}\rightarrow\Phi\left(R_{wrt}\left(A(x)\right)\right) \qquad [\Phi \text{ Rule 24]}$$

where T_c is the type of channel c. □

Link Rule 37 (Input Channel). For the simplified input prefixing $c?x \rightarrow A(x)$, it is equal to restricted input prefixing with the predicate $P = \mathbf{True}$ [35, Definition B.16].

$$\Upsilon\left(c?x\rightarrow A(x)\right)$$

$$\Upsilon\left(c?x : \mathbf{True}\rightarrow A(x)\right)$$

$$= c?x : \{y \mid y\text{<-}\Phi(T_c), \Phi\left(\mathbf{True}\right)\}\rightarrow\Phi\left(R_{wrt}\left(A(x)\right)\right) \qquad \text{[Link Rule 36]}$$

$$= c?x : \{y \mid y\text{<-}\Phi(T_c), \mathit{true}\}\rightarrow\Phi\left(R_{wrt}\left(A(x)\right)\right) \qquad [\Phi \text{ Rule 1 and Table D.3]}$$

$$= c?x : \{y \mid y\text{<-}\Phi(T_c)\}\rightarrow\Phi\left(R_{wrt}\left(A(x)\right)\right) \qquad \text{[Simplified set comprehension]}$$

$$= c?x : \Phi(T_c)\rightarrow\Phi\left(R_{wrt}\left(A(x)\right)\right)$$

$$\text{[Taking all elements from one set to form a new set is equal to this set itself]}$$

$$= c?x\rightarrow\Phi\left(R_{wrt}\left(A(x)\right)\right) \qquad [T_c \text{ is the type of } c \text{ and Link Rule 2]}$$

□

Multiple Data Transfer Channel

Link Rule 38 (Multiple Data Transfer Channel). For the channel with multiple inputs, or outputs, or the combination of them, the rule is a combination of corresponding rules as well. □

E.8.2.3 Guarded Action

Link Rule 39 (Guarded Action).

$$
\begin{aligned}
&\Upsilon\left((g)\mathbin{\&}A\right)\\
&=\begin{cases}\Omega\left(R_{wrt}\left((g)\mathbin{\&}A\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left((g)\mathbin{\&}A\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&=\Phi\left(R_{wrt}\left((g)\mathbin{\&}A\right)\right) && \text{[Only behavioural part]}\\
&=\Phi\left(R_{mrg}\left(R_{pre}\left(g\right),R_{pre}\left(A\right)\right)\rightarrow\left((g)\mathbin{\&}R_{post}\left(A\right)\right)\right) && [R_{wrt}\text{ Rule 28}]\\
&=\Phi\left(R_{mrg}\left(R_{pre}\left(g\right),R_{pre}\left(A\right)\right)\right)\rightarrow\ \Phi\left((g)\mathbin{\&}R_{post}\left(A\right)\right) && [\Phi\text{ Rule 24}]\\
&=\Phi\left(R_{mrg}\left(R_{pre}\left(g\right),R_{pre}\left(A\right)\right)\right)\rightarrow\ \Phi\left(g\right)\ \mathbin{\&}\Phi\left(R_{post}\left(A\right)\right) && [\Phi\text{ Rule 25}]
\end{aligned}
$$

Provided g evaluates the state variables $s_g = s_{gi}, \cdots, s_{gj}$ and the initial construct of A evaluates $s_a = s_{ai}, \cdots, s_{aj}$. All state variables evaluated in either g or the initial construct of A are the union of s_g and s_a, denoted as $s_i, \cdots, s_j$. Then

$$
\begin{aligned}
&\Upsilon\left((g)\mathbin{\&}A\right)\\
&=\Phi\left(R_{mrg}\left(R_{pre}\left(g\right),R_{pre}\left(A\right)\right)\right)\rightarrow\ \Phi\left(g\right)\ \mathbin{\&}\Phi\left(R_{post}\left(A\right)\right) && []\\
&=\Phi\left(R_{mrg}\left(\begin{array}{l}\left(P\_Op\_s_{gi}\right)\rightarrow\cdots\rightarrow\left(P\_Op\_s_{gj}\right),\\ \left(P\_Op\_s_{ai}\right)\rightarrow\cdots\rightarrow\left(P\_Op\_s_{aj}\right)\end{array}\right)\right)\rightarrow\ \Phi\left(g\right)\ \mathbin{\&}\Phi\left(R_{post}\left(A\right)\right)\\
& && \text{[Definition 4.3.1]}\\
&=\Phi\left(\left(P\_Op\_s_i\right)\rightarrow\cdots\rightarrow\left(P\_Op\_s_j\right)\right)\rightarrow\ \Phi\left(g\right)\ \mathbin{\&}\Phi\left(R_{post}\left(A\right)\right)\\
& && \text{[Definition 4.3.2]}\\
&=\Phi\left(\left(P\_Op\_s_i\right)\right)\rightarrow\cdots\rightarrow\Phi\left(\left(P\_Op\_s_j\right)\right)\rightarrow\ \Phi\left(g\right)\ \mathbin{\&}\Phi\left(R_{post}\left(A\right)\right)\\
& && [\Phi\text{ Rule 24}]\\
&=\begin{cases}\mathit{channel}\ P\_Op\_s_i : \Phi(T_{s_i})\\ \cdots\\ \mathit{channel}\ P\_Op\_s_j : \Phi(T_{s_j})\\ \mathit{HIDE\_CSPB} = \{\!|P\_Op\_s_i, \cdots, P\_Op\_s_j|\!\}\\ P\_Op\_s_i?s_i\rightarrow\cdots\rightarrow P\_Op\_s_j?s_j\rightarrow\Phi\left(g\right)\ \mathbin{\&}\Phi\left(R_{post}\left(A\right)\right)\end{cases} && [\Phi\text{ Rule 22}]
\end{aligned}
$$

□

E.8.2.4 Sequential Composition

Link Rule 40 (Sequential Composition).

$$
\begin{aligned}
&\Upsilon\left(A_1\ ;A_2\right)\\
&=\begin{cases}\Omega\left(R_{wrt}\left(A_1\ ;A_2\right)\right) & \text{State Part}\\ \\ \Phi\left(R_{wrt}\left(A_1\ ;A_2\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&=\Phi\left(R_{wrt}\left(A_1\ ;A_2\right)\right) && \text{[Only behavioural part in this construct]}\\
&=\Phi\left(R_{pre}\left(A_1\right)\rightarrow\left(R_{post}\left(A_1\right);R_{wrt}\left(A_2\right)\right)\right) && [R_{wrt}\text{ Rule 29}]\\
&=\Phi\left(R_{pre}\left(A_1\right)\right)\rightarrow\Phi\left(R_{post}\left(A_1\right);R_{wrt}\left(A_2\right)\right) && [\Phi\text{ Rule 24}]\\
&=\Phi\left(R_{pre}\left(A_1\right)\right)\rightarrow\left(\Phi\left(R_{post}\left(A_1\right)\right);\Phi\left(R_{wrt}\left(A_2\right)\right)\right) && [\Phi\text{ Rule 26}]
\end{aligned}
$$

□

E.8.2.5 External Choice

Link Rule 41 (External Choice).

$$\begin{aligned}
&\Upsilon\left(A_1 \Box A_2\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(A_1 \Box A_2\right)\right) & \text{State Part} \\ & \\ \Phi\left(R_{wrt}\left(A_1 \Box A_2\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\left(R_{wrt}\left(A_1 \Box A_2\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right) \rightarrow \left(R_{post}\left(A_1\right) \Box R_{post}\left(A_2\right)\right)\right) && [R_{wrt}\text{ Rule 30]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \Phi\left(R_{post}\left(A_1\right) \Box R_{post}\left(A_2\right)\right) && [\Phi\text{ Rule 24]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(A_1\right)\right) \Box \Phi\left(R_{post}\left(A_2\right)\right)\right)\\
& && [\Phi\text{ Rule 27]}
\end{aligned}$$

where A_1 and A_2 are limited to AA actions—prefixed actions defined in Definition B.3.1.

□

E.8.2.6 Internal Choice

Link Rule 42 (Internal Choice).

$$\begin{aligned}
&\Upsilon\left(A_1 \sqcap A_2\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(A_1 \sqcap A_2\right)\right) & \text{State Part} \\ & \\ \Phi\left(R_{wrt}\left(A_1 \sqcap A_2\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\left(R_{wrt}\left(A_1 \sqcap A_2\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right) \rightarrow \left(R_{post}\left(A_1\right) \sqcap R_{post}\left(A_2\right)\right)\right) && [R_{wrt}\text{ Rule 31]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \Phi\left(R_{post}\left(A_1\right) \sqcap R_{post}\left(A_2\right)\right) && [\Phi\text{ Rule 24]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \left(\Phi\left(R_{post}\left(A_1\right)\right) \sqcap \Phi\left(R_{post}\left(A_2\right)\right)\right)\\
& && [\Phi\text{ Rule 29]}
\end{aligned}$$

□

E.8.2.7 Parallel Composition

Link Rule 43 (Parallel Composition (Disjoint Variables in Scope)).

$$\begin{aligned}
&\Upsilon\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right)\right) & \text{State Part} \\ & \\ \Phi\left(R_{wrt}\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\left(R_{wrt}\left(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}(A_1), R_{pre}(A_2)\right) \rightarrow \left(R_{post}\left(A_1\right) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket R_{post}\left(A_2\right)\right)\right)\\
& && [R_{wrt}\text{ Rule 32]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \Phi\left(R_{post}\left(A_1\right) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket R_{post}\left(A_2\right)\right)\\
& && [\Phi\text{ Rule 24]}\\
&= \Phi\left(R_{mrg}\left(R_{pre}\left(A_1\right), R_{pre}\left(A_2\right)\right)\right) \rightarrow \begin{pmatrix} \Phi\left(R_{post}\left(A_1\right)\right) \\ \underset{\Phi(cs)}{\|} \\ \Phi\left(R_{post}\left(A_2\right)\right)\end{pmatrix} && [\Phi\text{ Rule 30]}
\end{aligned}$$

provided

$$ns_1 = scpV\,(A_1)$$
$$ns_2 = scpV\,(A_2)$$

□

Link Rule 44 (Parallel Composition (Disjoint Variables in Updating)).

$$\begin{aligned}
&\Upsilon\,(A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)\\
&= \begin{cases} \Omega\,(R_{wrt}\,(A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)) & \text{State Part}\\ \\ \Phi\,(R_{wrt}\,(A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\,(R_{wrt}\,(A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)) && \text{[Only behavioural part in this construct]}\\
&= \Phi\,(R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to (R_{post}\,(A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, R_{post}\,(A_2)))\\
& && [R_{wrt} \text{ Rule 32}]\\
&= \Phi\,(R_{mrg}\,(R_{pre}\,(A_1)\,, R_{pre}\,(A_2))) \to \Phi\,(R_{post}\,(A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, R_{post}\,(A_2))\\
& && [\Phi \text{ Rule 24}]\\
&= \Phi\,(R_{mrg}\,(R_{pre}\,(A_1)\,, R_{pre}\,(A_2))) \to \begin{pmatrix} \Phi\,(R_{post}\,(A_1)) \\ \underset{\Phi(cs)}{\|} \\ \Phi\,(R_{post}\,(A_2)) \end{pmatrix} && [\Phi \text{ Rule 31}]
\end{aligned}$$

provided

$$wrtV\,(A_1) = ns_1$$
$$wrtV\,(A_2) = ns_2$$
$$wrtV\,(A_1) \cap scpV\,(A_2) = \varnothing$$
$$wrtV\,(A_2) \cap scpV\,(A_1) = \varnothing$$

□

E.8.2.8 Interleaving

Link Rule 45 (Interleaving (Disjoint Variables in Scope)).

$$\begin{aligned}
&\Upsilon\,(A_1 \,|\!|\!\![\, ns_1 \mid ns_2 \,]\!\!|\!|\, A_2)\\
&= \begin{cases} \Omega\,(R_{wrt}\,(A_1 \,|\!|\!\![\, ns_1 \mid ns_2 \,]\!\!|\!|\, A_2)) & \text{State Part}\\ \\ \Phi\,(R_{wrt}\,(A_1 \,|\!|\!\![\, ns_1 \mid ns_2 \,]\!\!|\!|\, A_2)) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\,(R_{wrt}\,(A_1 \,|\!|\!\![\, ns_1 \mid ns_2 \,]\!\!|\!|\, A_2)) && \text{[Only behavioural part in this construct]}\\
&= \Phi\,(R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to (R_{post}\,(A_1) \,|\!|\!\![\, ns_1 \mid ns_2 \,]\!\!|\!|\, R_{post}\,(A_2)))\\
& && [R_{wrt} \text{ Rule 32}]\\
&= \Phi\,(R_{mrg}\,(R_{pre}\,(A_1)\,, R_{pre}\,(A_2))) \to \Phi\,(R_{post}\,(A_1) \,|\!|\!\![\, ns_1 \mid ns_2 \,]\!\!|\!|\, R_{post}\,(A_2))\\
& && [\Phi \text{ Rule 24}]\\
&= \Phi\,(R_{mrg}\,(R_{pre}\,(A_1)\,, R_{pre}\,(A_2))) \to \begin{pmatrix} \Phi\,(R_{post}\,(A_1)) \\ ||| \\ \Phi\,(R_{post}\,(A_2)) \end{pmatrix} && [\Phi \text{ Rule 30}]
\end{aligned}$$

provided

$$ns_1 = scpV\,(A_1)$$
$$ns_2 = scpV\,(A_2)$$

□

Link Rule 46 (Interleaving (Disjoint Variables in Updating)).

$$
\begin{aligned}
&\Upsilon\,(A_1 \mathbin{|\!|\![}\, ns_1 \mid ns_2 \,]\!|\!|\, A_2)\\
&= \begin{cases} \Omega\,(R_{wrt}\,(A_1 \mathbin{|\!|\![}\, ns_1 \mid ns_2 \,]\!|\!|\, A_2)) & \text{State Part}\\[1em] \Phi\,(R_{wrt}\,(A_1 \mathbin{|\!|\![}\, ns_1 \mid ns_2 \,]\!|\!|\, A_2)) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\,(R_{wrt}\,(A_1 \mathbin{|\!|\![}\, ns_1 \mid ns_2 \,]\!|\!|\, A_2)) && \text{[Only behavioural part in this construct]}\\
&= \Phi\,(R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to (R_{post}\,(A_1) \mathbin{|\!|\![}\, ns_1 \mid ns_2 \,]\!|\!|\, R_{post}\,(A_2)))\\
& && [R_{wrt}\ \text{Rule 32}]\\
&= \Phi\,(R_{mrg}\,(R_{pre}\,(A_1)\,, R_{pre}\,(A_2))) \to \Phi\,(R_{post}\,(A_1) \mathbin{|\!|\![}\, ns_1 \mid ns_2 \,]\!|\!|\, R_{post}\,(A_2))\\
& && [\Phi\ \text{Rule 24}]\\
&= \Phi\,(R_{mrg}\,(R_{pre}\,(A_1)\,, R_{pre}\,(A_2))) \to \left(\begin{array}{l} \Phi\,(R_{post}\,(A_1))\\ |\!|\!|\\ \Phi\,(R_{post}\,(A_2))\end{array}\right) && [\Phi\ \text{Rule 30}]
\end{aligned}
$$

provided

$$
\begin{aligned}
&wrtV\,(A_1) = ns_1\\
&wrtV\,(A_2) = ns_2\\
&wrtV\,(A_1) \cap scpV\,(A_2) = \varnothing\\
&wrtV\,(A_2) \cap scpV\,(A_1) = \varnothing
\end{aligned}
$$

□

E.8.2.9 Hiding

Link Rule 47 (Hiding).

$$
\begin{aligned}
&\Upsilon\,(A \setminus cs)\\
&= \begin{cases} \Omega\,(R_{wrt}\,(A \setminus cs)) & \text{State Part}\\[1em] \Phi\,(R_{wrt}\,(A \setminus cs)) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\,(R_{wrt}\,(A \setminus cs)) && \text{[Only behavioural part in this construct]}\\
&= \Phi\,(R_{Pre}\,(A) \to (R_{Post}\,(A) \setminus cs)) && [R_{wrt}\ \text{Rule 33}]\\
&= \Phi\,(R_{Pre}\,(A)) \to \Phi\,(R_{Post}\,(A) \setminus cs) && [\Phi\ \text{Rule 24}]\\
&= \Phi\,(R_{Pre}\,(A)) \to (\Phi\,(R_{Post}\,(A)) \setminus \Phi\,(cs)) && [\Phi\ \text{Rule 33}]
\end{aligned}
$$

□

E.8.2.10 Recursion

Link Rule 48 (Recursion).

$$
\begin{aligned}
&\Upsilon\,(\mu X \bullet A\,(X))\\
&= \begin{cases} \Omega\,(R_{wrt}\,(\mu X \bullet A\,(X))) & \text{State Part}\\[1em] \Phi\,(R_{wrt}\,(\mu X \bullet A\,(X))) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\,(R_{wrt}\,(\mu X \bullet A\,(X))) && \text{[Only behavioural part in this construct]}\\
&= \Phi\,(\mu X \bullet R_{wrt}\,(A(X))) && [R_{wrt}\ \text{Rule 34}]\\
&= \text{let X} = \Phi\,(R_{wrt}\,(A(X)))\ \ \text{within X} && [\Phi\ \text{Rule 34}]
\end{aligned}
$$

□

E.8.2.11 Iterated Sequential Composition

Link Rule 49 (Iterated Sequential Composition).

$$
\begin{aligned}
&\Upsilon\left(;\, x : T \bullet A(x)\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(;\, x : T \bullet A(x)\right)\right) & \text{State Part}\\ \Phi\left(R_{wrt}\left(;\, x : T \bullet A(x)\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\left(R_{wrt}\left(;\, x : T \bullet A(x)\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{pre}\left(A(x)\right) \rightarrow \left(;\, x : T \bullet R_{post}\left(A(x)\right)\right)\right) && [R_{wrt} \text{ Rule 38}]\\
&= \Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \Phi\left(;\, x : T \bullet R_{post}\left(A(x)\right)\right) && [\Phi \text{ Rule 24}]\\
&= \Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow ;_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right) && [\Phi \text{ Rule 35}]
\end{aligned}
$$

□

E.8.2.12 Iterated External Choice

Link Rule 50 (Iterated External Choice).

$$
\begin{aligned}
&\Upsilon\left(\Box\, x : T \bullet A(x)\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(\Box\, x : T \bullet A(x)\right)\right) & \text{State Part}\\ \Phi\left(R_{wrt}\left(\Box\, x : T \bullet A(x)\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\left(R_{wrt}\left(\Box\, x : T \bullet A(x)\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{pre}\left(A(x)\right) \rightarrow \left(\Box\, x : T \bullet R_{post}\left(A(x)\right)\right)\right) && [R_{wrt} \text{ Rule 38}]\\
&= \Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \Phi\left(\Box\, x : T \bullet R_{post}\left(A(x)\right)\right) && [R_{wrt} \text{ Rule 24}]\\
&= \Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \Box_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right) && [\Phi \text{ Rule 36}]
\end{aligned}
$$

where AA is a prefixed action defined in Definition B.3.1. □

E.8.2.13 Iterated Internal Choice

Link Rule 51 (Iterated Internal Choice).

$$
\begin{aligned}
&\Upsilon\left(\sqcap\, x : T \bullet A(x)\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(\sqcap\, x : T \bullet A(x)\right)\right) & \text{State Part}\\ \Phi\left(R_{wrt}\left(\sqcap\, x : T \bullet A(x)\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \Phi\left(R_{wrt}\left(\sqcap\, x : T \bullet A(x)\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{pre}\left(A(x)\right) \rightarrow \Phi\left(\sqcap\, x : T \bullet R_{post}\left(A(x)\right)\right)\right) && [R_{wrt} \text{ Rule 38}]\\
&= \Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \Phi\left(\sqcap\, x : T \bullet R_{post}\left(A(x)\right)\right) && [\Phi \text{ Rule 24}]\\
&= \Phi\left(R_{pre}\left(A(x)\right)\right) \rightarrow \sqcap_{x:\Phi(T)} \bullet \Phi\left(R_{post}\left(A(x)\right)\right) && [\Phi \text{ Rule 37}]
\end{aligned}
$$

□

E.8.2.14 Iterated Parallel Composition

The iterated parallel composition of actions is not supported yet.

E.8.2.15 Iterated Interleaving

The iterated interleaving of actions is not supported yet.

E.8.3 Command

E.8.3.1 Assignment

Link Rule 52 (Assignment). Provided $e_{s_i}, \cdots, e_{s_j}, e_{l_k}, \cdots, e_{l_m}$ evaluate local variables $l_p, \cdots, l_q$, then

$$
\begin{array}{l}
\Upsilon\left(s_i, \cdots, s_j, l_k, \cdots, l_m := e_{s_i}, \cdots, e_{s_j}, e_{l_k}, \cdots, e_{l_m}\right) \\
= \left\{\begin{array}{ll}
\Omega\left(R_{wrt}\left(s_i, \cdots, s_j, l_k, \cdots, l_m := e_{s_i}, \cdots, e_{s_j}, e_{l_k}, \cdots, e_{l_m}\right)\right) & \text{State Part} \\
\Phi\left(R_{wrt}\left(s_i, \cdots, s_j, l_k, \cdots, l_m := e_{s_i}, \cdots, e_{s_j}, e_{l_k}, \cdots, e_{l_m}\right)\right) & \text{Behaviour Part}
\end{array}\right. \\
\hfill [\text{Link Definition}] \\
= \left\{\begin{array}{ll}
\Omega\left(\begin{array}{l}
P\_assOp == \\
\left[\begin{array}{l}
\Delta P\_StPar \,;\, l_p? : T_{l_p} \,;\, \cdots \,;\, l_q? : T_{l_q}; \\
\quad l_k! : T_{l_k} \,;\, \cdots \,;\, l_m! : T_{l_m} \mid \\
P\_s_i' = e_{s_i}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \cdots \wedge \\
\quad P\_s_j' = e_{s_j}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \\
l_k! = e_{l_k}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge \cdots \wedge \\
\quad l_m! = e_{l_m}[l_p?, \cdots, l_q?/l_p, \cdots, l_q] \wedge u' = u
\end{array}\right]
\end{array}\right) & \text{State Part} \\
\Phi\left(\left(P\_assOp\right)\right) & \text{Behaviour Part}
\end{array}\right. \\
\hfill [R_{wrt} \text{ Rule 24 and } R_{wrt} \text{ Rule 39}] \\
= \left\{\begin{array}{ll}
\Omega_3\left(\Omega_2\left(\ P\_assOp == \cdots\ \right)\right) & \text{State Part} \\
\left(\begin{array}{l}
\mathit{channel}\ P\_assOp : \\
\quad \Phi\left(T_{l_p}\right).\cdots.\Phi\left(T_{l_q}\right).\Phi\left(T_{l_k}\right).\cdots.\Phi\left(T_{l_m}\right) \\
HIDE\_CSPB = \{\!| P\_assOp |\!\} \\
(P\_assOp!l_p!\cdots!l_q?l_k?\cdots?l_m \rightarrow SKIP)
\end{array}\right) & \text{Behaviour Part}
\end{array}\right. \\
\hfill [\Omega_1 \text{ Rule 1 and } \Phi \text{ Rule 22}] \\
= \left\{\begin{array}{ll}
\Omega_3\left(\ P\_assOp \mathrel{\widehat=} \cdots\ \right) & \text{State Part} \\
\left(\begin{array}{l}
\mathit{channel}\ P\_assOp : \\
\quad \Phi\left(T_{l_p}\right).\cdots.\Phi\left(T_{l_q}\right).\Phi\left(T_{l_k}\right).\cdots.\Phi\left(T_{l_m}\right) \\
HIDE\_CSPB = \{\!| P\_assOp |\!\} \\
(P\_assOp!l_p!\cdots!l_q?l_k?\cdots?l_m \rightarrow SKIP)
\end{array}\right) & \text{Behaviour Part}
\end{array}\right. \\
\hfill [\Omega_2 \text{ Rule 4}]
\end{array}
$$

The precondition of P_assOp is always *true*. □

E.8.3.2 Alternation

Link Rule 53 (Alternation). Provided $g_1, \cdots, g_n$ and the first events of $A_1, \cdots, A_n$ evaluate state variables $s_i, \cdots, s_j$, then

$$
\Upsilon\left(\begin{array}{ll}
\textbf{if} & g_1 \longrightarrow A_1 \\
\square & g_2 \longrightarrow A_2 \\
\square & \ldots \\
\square & g_n \longrightarrow A_n \\
\textbf{fi} &
\end{array}\right)
$$

$$= \left\{ \begin{array}{l} \Omega \left(R_{wrt} \left(\begin{array}{ll} \textbf{if} & g_1 \longrightarrow A_1 \\ [\!] & g_2 \longrightarrow A_2 \\ [\!] & \ldots \\ [\!] & g_n \longrightarrow A_n \\ \textbf{fi} & \end{array} \right) \right) \text{ State Part} \\ \Phi \left(R_{wrt} \left(\begin{array}{ll} \textbf{if} & g_1 \longrightarrow A_1 \\ [\!] & g_2 \longrightarrow A_2 \\ [\!] & \ldots \\ [\!] & g_n \longrightarrow A_n \\ \textbf{fi} & \end{array} \right) \right) \text{ Behaviour Part} \end{array} \right. \qquad \text{[Link Definition]}$$

$$= \Phi \left(R_{wrt} \left(\begin{array}{ll} \textbf{if} & g_1 \longrightarrow A_1 \\ [\!] & g_2 \longrightarrow A_2 \\ [\!] & \ldots \\ [\!] & g_n \longrightarrow A_n \\ \textbf{fi} & \end{array} \right) \right) \qquad \text{[Only behavioural part in this construct]}$$

$$= \Phi \left(R_{wrt} \left(\begin{array}{ll} & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \textbf{Chaos} \\ \Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; A_1 \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \;\&\; (A_1 \sqcap A_2) \\ \Box & \ldots \\ \Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \;\&\; (A_i \sqcap A_j \sqcap A_k) \\ \Box & \ldots \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \;\&\; (A_1 \sqcap A_2 \sqcap \cdots \sqcap g_n) \end{array} \right) \right)$$

[R_{wrt} Rule 40]

$$= \quad \Phi \left(\begin{array}{l} R_{mrg} \left(\begin{array}{l} R_{pre} \left((\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \textbf{Chaos} \right), \\ \cdots, \\ R_{pre} \left((g_1 \wedge g_2 \wedge \cdots \wedge g_n) \;\&\; (A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n) \right) \end{array} \right) \to \\ \left(\begin{array}{ll} & R_{post} \left((\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; \textbf{Chaos} \right) \\ \Box & R_{post} \left((g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; A_1 \right) \\ \Box & R_{post} \left((g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \;\&\; (A_1 \sqcap A_2) \right) \\ \Box & \ldots \\ \Box & R_{post} \left((\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \;\&\; (A_i \sqcap A_j \sqcap A_k) \right) \\ \Box & \ldots \\ \Box & R_{post} \left((g_1 \wedge g_2 \wedge \cdots \wedge g_n) \;\&\; (A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n) \right) \end{array} \right) \end{array} \right)$$

[External choice R_{wrt} Rule 30]

$$= \quad \Phi \left(\begin{array}{l} R_{mrg} \left(\begin{array}{l} R_{mrg} \left(\; R_{pre} \left((\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \right), R_{pre} \left(\textbf{Chaos} \right) \; \right), \\ \cdots, \\ R_{mrg} \left(\begin{array}{l} R_{pre} \left((g_1 \wedge g_2 \wedge \cdots \wedge g_n) \right), \\ R_{pre} \left(A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n \right) \end{array} \right) \end{array} \right) \to \\ \left(\begin{array}{ll} & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; R_{post} \left(\textbf{Chaos} \right) \\ \Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \;\&\; R_{post} \left(A_1 \right) \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \;\&\; R_{post} \left(A_1 \sqcap A_2 \right) \\ \Box & \ldots \\ \Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \;\&\; R_{post} \left(A_i \sqcap A_j \sqcap A_k \right) \\ \Box & \ldots \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \;\&\; R_{post} \left(A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n \right) \end{array} \right) \end{array} \right)$$

[Guarded action R_{wrt} Rule 28]

$$
= \quad \Phi \left(\begin{array}{l}
R_{mrg} \left(\begin{array}{l}
R_{mrg} \left(\begin{array}{l} R_{pre}(g_1) \to R_{pre}(g_2) \to \cdots \to R_{pre}(g_n), \\ R_{pre}(\mathbf{Chaos}) \end{array} \right), \\
\cdots, \\
R_{mrg} \left(\begin{array}{l} R_{pre}(g_1) \to R_{pre}(g_2) \to \cdots \to R_{pre}(g_n), \\ R_{pre}(A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n) \end{array} \right)
\end{array} \right) \to \\
\left(\begin{array}{ll}
 & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \,\&\, R_{post}(\mathbf{Chaos}) \\
\Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \,\&\, R_{post}(A_1) \\
\Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \,\&\, R_{post}(A_1 \sqcap A_2) \\
\Box & \ldots \\
\Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \,\&\, R_{post}(A_i \sqcap A_j \sqcap A_k) \\
\Box & \ldots \\
\Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \,\&\, R_{post}(A_1 \sqcap A_2 \sqcap \cdots \sqcap A_n)
\end{array} \right)
\end{array} \right)
$$

[Definition 4.3.1]

$$
= \quad \Phi \left(\begin{array}{l}
R_{mrg} \left(\begin{array}{l}
R_{mrg} \left(\begin{array}{l} R_{pre}(g_1) \to R_{pre}(g_2) \to \cdots \to R_{pre}(g_n), \\ R_{pre}(\mathbf{Chaos}) \end{array} \right), \\
\cdots, \\
R_{mrg} \left(\begin{array}{l} R_{pre}(g_1) \to R_{pre}(g_2) \to \cdots \wedge R_{pre}(g_n), \\ R_{mrg}(R_{pre}(A_1), R_{pre}(A_2), \cdots, R_{pre}(A_n)) \end{array} \right)
\end{array} \right) \to \\
\left(\begin{array}{ll}
 & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \,\&\, R_{post}(\mathbf{Chaos}) \\
\Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \,\&\, R_{post}(A_1) \\
\Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \,\&\, (R_{post}(A_1) \sqcap R_{post}(A_2)) \\
\Box & \ldots \\
\Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \,\&\, \left(\begin{array}{ll} & R_{post}(A_i) \\ \sqcap & R_{post}(A_j) \\ \sqcap & R_{post}(A_k) \end{array} \right) \\
\Box & \ldots \\
\Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \,\&\, \left(\begin{array}{ll} & R_{post}(A_1) \\ \sqcap & R_{post}(A_2) \\ \sqcap & \cdots \\ \sqcap & R_{post}(A_n) \end{array} \right)
\end{array} \right)
\end{array} \right)
$$

[Internal choice R_{wrt} Rule 31]

$$
= \quad \Phi \left(\begin{array}{l}
R_{mrg} \left(\begin{array}{l} R_{pre}(g_1), R_{pre}(g_2), \cdots, R_{pre}(g_n), \\ R_{pre}(A_1), R_{pre}(A_2), \cdots, R_{pre}(A_n) \end{array} \right) \to \\
\left(\begin{array}{ll}
 & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \,\&\, \mathbf{Chaos} \\
\Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \,\&\, R_{post}(A_1) \\
\Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \,\&\, (R_{post}(A_1) \sqcap R_{post}(A_2)) \\
\Box & \ldots \\
\Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \,\&\, \left(\begin{array}{ll} & R_{post}(A_i) \\ \sqcap & R_{post}(A_j) \\ \sqcap & R_{post}(A_k) \end{array} \right) \\
\Box & \ldots \\
\Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \,\&\, \left(\begin{array}{ll} & R_{post}(A_1) \\ \sqcap & R_{post}(A_2) \\ \sqcap & \cdots \\ \sqcap & R_{post}(A_n) \end{array} \right)
\end{array} \right)
\end{array} \right)
$$

[Definition 4.3.2 and R_{wrt} Rule 26]

$$= \Phi \left(\begin{array}{l} (P\_Op\_s_i) \rightarrow \cdots \rightarrow (P\_Op\_s_j) \rightarrow \\ \left(\begin{array}{ll} & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} \mathbf{Chaos} \\ \Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} R_{post}(A_1) \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} (R_{post}(A_1) \sqcap R_{post}(A_2)) \\ \Box & \ldots \\ \Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \mathbin{\&} \left(\begin{array}{ll} & R_{post}(A_i) \\ \sqcap & R_{post}(A_j) \\ \sqcap & R_{post}(A_k) \end{array} \right) \\ \Box & \ldots \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \mathbin{\&} \left(\begin{array}{ll} & R_{post}(A_1) \\ \sqcap & R_{post}(A_2) \\ \sqcap & \cdots \\ \sqcap & R_{post}(A_n) \end{array} \right) \end{array} \right) \end{array} \right)$$

[Assumption, Definition 4.3.2 and Definition 4.3.1]

$$= \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \Phi \left(\begin{array}{ll} & (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} \mathbf{Chaos} \\ \Box & (g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} R_{post}(A_1) \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} (R_{post}(A_1) \sqcap R_{post}(A_2)) \\ \Box & \ldots \\ \Box & (\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \mathbin{\&} \left(\begin{array}{ll} & R_{post}(A_i) \\ \sqcap & R_{post}(A_j) \\ \sqcap & R_{post}(A_k) \end{array} \right) \\ \Box & \ldots \\ \Box & (g_1 \wedge g_2 \wedge \cdots \wedge g_n) \mathbin{\&} \left(\begin{array}{ll} & R_{post}(A_1) \\ \sqcap & R_{post}(A_2) \\ \sqcap & \cdots \\ \sqcap & R_{post}(A_n) \end{array} \right) \end{array} \right) \end{array} \right)$$

[Φ Rule 24 and Φ Rule 22]

$$= \left(\begin{array}{l} Op\_s_i?s_i \rightarrow \cdots \rightarrow Op\_s_j?s_j \rightarrow \\ \left(\begin{array}{ll} & \Phi(\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} \mathbf{div} \\ \Box & \Phi(g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} \Phi(R_{post}(A_1)) \\ \Box & \Phi(g_1 \wedge g_2 \wedge \cdots \wedge \neg g_n) \mathbin{\&} (\Phi(R_{post}(A_1)) \sqcap \Phi(R_{post}(A_2))) \\ \Box & \ldots \\ \Box & \Phi(\cdots \wedge g_i \cdots \wedge g_j \wedge \cdots \wedge g_k \wedge \cdots) \mathbin{\&} \left(\begin{array}{ll} & \Phi(R_{post}(A_i)) \\ \sqcap & \Phi(R_{post}(A_j)) \\ \sqcap & \Phi(R_{post}(A_k)) \end{array} \right) \\ \Box & \ldots \\ \Box & \Phi(g_1 \wedge g_2 \wedge \cdots \wedge g_n) \mathbin{\&} \left(\begin{array}{ll} & \Phi(R_{post}(A_1)) \\ \sqcap & \Phi(R_{post}(A_2)) \\ \sqcap & \cdots \\ \sqcap & \Phi(R_{post}(A_n)) \end{array} \right) \end{array} \right) \end{array} \right)$$

[Φ Rule 27, Φ Rule 25, Φ Rule 29 and Φ Rule 23]

□

E.8.3.3 Variable Block

Link Rule 54 (Variable Block).

$$\Upsilon(\mathbf{var}\, x : T \bullet A)$$

$$= \begin{cases} \Omega(R_{wrt}(\mathbf{var}\, x : T \bullet A)) & \text{State Part} \\ \Phi(R_{wrt}(\mathbf{var}\, x : T \bullet A)) & \text{Behaviour Part} \end{cases}$$

[Link Definition]

$$
\begin{aligned}
&= \Phi\left(R_{wrt}\left(\mathbf{var}\; x : T \bullet A\right)\right) && \text{[Only behavioural part in this construct]}\\
&= \Phi\left(R_{pre}\left(A\right) \rightarrow \left(\mathbf{var}\; x : T \bullet R_{post}\left(A\right)\right)\right) && [R_{wrt} \text{ Rule 41}]\\
&= \Phi\left(R_{pre}\left(A\right)\right) \rightarrow \Phi\left(\mathbf{var}\; x : T \bullet R_{post}\left(A\right)\right) && [\Phi \text{ Rule 24}]\\
&= \Phi\left(R_{pre}\left(A\right)\right) \rightarrow \sqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}(A)\right), \{x\}\right) && [\Phi \text{ Rule 38}]
\end{aligned}
$$

□

E.8.3.4 Specification Statement

Link Rule 55 (Specification Statement). Provided

- the frame w is composed of state variables s_w and local variables l_w
- *pre* and *post* contain free occurrences of before-state local variables l_b
- *post* contains free occurrences of after-state local variables l'_a
- u denotes the variables whose dashed version occurs in *post* but the undashed version is not in the frame
- s_u denotes all state variables that are not in the s_w.

then

$$
\begin{aligned}
&\Upsilon\left(w : [\,pre, post\,]\right)\\
&= \begin{cases} \Omega\left(R_{wrt}\left(w : [\,pre, post\,]\right)\right) & \text{State Part}\\ \Phi\left(R_{wrt}\left(w : [\,pre, post\,]\right)\right) & \text{Behaviour Part}\end{cases} && \text{[Link Definition]}\\
&= \begin{cases} \Omega\left(specOp == \left[\begin{array}{c} \Delta P\_StPar\,;\, l_b? : T_{l_b}\,;\, l_a! : T_{l_a} \mid pre[l_b?/l_b] \wedge \\ \exists\, u' : T_u \bullet post[l_b?/l_b, l_a!/l'_a] \wedge s'_u = s_u \end{array}\right]\right)\\ \\ \Phi\left(\left(specOp\right)\right) \end{cases}\\
& && [R_{wrt} \text{ Rule 43}]\\
&= \begin{cases} \Omega_3\left(\Omega_2\left(\begin{array}{l} P\_specOp == \left[\begin{array}{l} \Delta P\_StPar\,;\, \Xi Q_1\_StPar\,;\, \cdots\,; \\ \Xi Q_n\_StPar\,;\, l_b? : T_{l_b}\,;\, l_a! : T_{l_a} \mid \\ pre[l_b?/l_b] \wedge s'_u = s_u \wedge \\ \exists\, u' : T_u \bullet post[l_b?/l_b, l_a!/l'_a] \end{array}\right] \\ \\ P\_specOp\_fOp == \left[\begin{array}{l} \Xi P\_StPar\,;\, \Xi Q_1\_StPar\,;\, \cdots\,; \\ \Xi Q_n\_StPar\,;\, l_b? : T_{l_b} \mid \\ \neg\, \mathbf{pre}\, P\_specOp \end{array}\right] \end{array}\right)\right) \\ \\ \left(\begin{array}{l} channel\;\; P\_specOp : \Phi\left(T_{l_b}\right).\Phi\left(T_{l_a}\right) \\ channel\;\; P\_fspecOp : \Phi\left(T_{l_b}\right) \\ HIDE\_CSPB = \{\!| P\_specOp, P\_fspecOp |\!\} \\ P\_specOp!l_b?l_a \rightarrow SKIP \;\square\; P\_specOp\_fOp!l_b \rightarrow \mathbf{div} \end{array}\right) \end{cases}\\
& && [\Omega_1 \text{ Rule 1 and } \Phi \text{ Rule 21}]
\end{aligned}
$$

$$= \left\{ \begin{array}{l} \Omega_3 \left(\begin{array}{l} P\_specOp \mathrel{\widehat=} \left[\begin{array}{l} \Delta P\_StPar \, ; \Xi Q_1\_StPar \, ; \cdots ; \Xi Q_n\_StPar ; \\ l_b ? : T_{l_b} \, ; l_a ! : T_{l_a} \mid \\ (pre[l_b?/l_b] \land \exists\, u' : T_u \bullet post[l_b?/l_b, l_a!/l_a']) \\ \land\, s_u' = s_u \end{array} \right] \\ \\ P\_specOp\_fOp \mathrel{\widehat=} \left[\begin{array}{l} \Xi P\_StPar \, ; \Xi Q_1\_StPar \, ; \cdots ; \Xi Q_n\_StPar ; \\ l_b ? : T_{l_b} \mid \neg \, \mathbf{pre}\, P\_specOp \end{array} \right] \end{array} \right) \\ \\ \left(\begin{array}{l} channel\ \ P\_specOp : \Phi\,(T_{l_b}) \, .\Phi\,(T_{l_a}) \\ channel\ \ P\_fspecOp : \Phi\,(T_{l_b}) \\ HIDE\_CSPB = \{\!| P\_specOp, P\_fspecOp |\!\} \\ P\_specOp!l_b?l_a \rightarrow SKIP \,\Box\, P\_specOp\_fOp!l_b \rightarrow \mathbf{div} \end{array} \right) \end{array} \right.$$

[Ω_2 Rule 4]

□

E.8.3.5 Assumption

Link Rule 56 (Assumption).

$$\Upsilon\,(\{\, pre \,\})$$

$$= \left\{ \begin{array}{ll} \Omega\,(R_{wrt}\,(\{\, pre \,\})) & \text{State Part} \\ \\ \Phi\,(R_{wrt}\,(\{\, pre \,\})) & \text{Behaviour Part} \end{array} \right. \qquad \text{[Link Definition]}$$

$$= \left\{ \begin{array}{l} \Omega\,(assmpOp == [\Xi P\_StPar \, ; l_b ? : T_{l_b} \mid pre[l_b?/l_b]\,]) \\ \\ \Phi\,\big(\big(assmpOp\big)\big) \end{array} \right. \qquad [R_{wrt} \text{ Rule 44}]$$

$$= \left\{ \begin{array}{l} \Omega_3 \left(\Omega_2 \left(\begin{array}{l} P\_assmpOp == \left[\begin{array}{l} \Xi P\_StPar \, ; \Xi Q_1\_StPar \, ; \cdots ; \\ \Xi Q_n\_StPar \, ; l_b ? : T_{l_b} \mid pre[l_b?/l_b] \end{array} \right] \\ \\ P\_assmpOp\_fOp == \left[\begin{array}{l} \Xi P\_StPar \, ; \Xi Q_1\_StPar \, ; \cdots ; \\ \Xi Q_n\_StPar \, ; l_b ? : T_{l_b} \mid \\ \neg \, \mathbf{pre}\, P\_assmpOp \end{array} \right] \end{array} \right) \right) \\ \\ \left(\begin{array}{l} channel\ \ P\_assmpOp : \Phi\,(T_{l_b}) \\ channel\ \ P\_assmpOp\_fOp : \Phi\,(T_{l_b}) \\ HIDE\_CSPB = \{\!| P\_assmpOp, P\_assmpOp\_fOp |\!\} \\ P\_assmpOp!l_b \rightarrow SKIP \,\Box\, P\_assmpOp\_fOp!l_b \rightarrow \mathbf{div} \end{array} \right) \end{array} \right.$$

[Ω_1 Rule 1 and Φ Rule 21]

$$= \left\{ \begin{array}{l} \Omega_3 \left(\begin{array}{l} P\_assmpOp \mathrel{\widehat=} \left[\begin{array}{l} \Xi P\_StPar \, ; \Xi Q_1\_StPar \, ; \cdots ; \Xi Q_n\_StPar ; \\ l_b ? : T_{l_b} \mid pre[l_b?/l_b] \end{array} \right] \\ \\ P\_assmpOp\_fOp \mathrel{\widehat=} \left[\begin{array}{l} \Xi P\_StPar \, ; \Xi Q_1\_StPar \, ; \cdots ; \Xi Q_n\_StPar ; \\ l_b ? : T_{l_b} \mid \neg \, \mathbf{pre}\, P\_assmpOp \end{array} \right] \end{array} \right) \\ \\ \left(\begin{array}{l} channel\ \ P\_assmpOp : \Phi\,(T_{l_b}) \\ channel\ \ P\_assmpOp\_fOp : \Phi\,(T_{l_b}) \\ HIDE\_CSPB = \{\!| P\_assmpOp, P\_assmpOp\_fOp |\!\} \\ P\_assmpOp!l_b \rightarrow SKIP \,\Box\, P\_assmpOp\_fOp!l_b \rightarrow \mathbf{div} \end{array} \right) \end{array} \right.$$

[Ω_2 Rule 4]

□

E.8.3.6 Coercion

Link Rule 57 (Coercion).

$$\Upsilon\left([\, post\,]\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left([\, post\,]\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left([\, post\,]\right)\right) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(coerOp == \left[\ \Xi P\_StPar\, ;\, l_b? : T_{l_b} \mid \exists\, u' : T_u \bullet post[l_b?/l_b]\ \right]\right) \\ \\ \Phi\left(\left(coerOp\right)\right) \end{cases}$$

$$[R_{wrt} \text{ Rule } 45]$$

$$= \begin{cases} \Omega_3\left(\Omega_2\left(\ P\_coerOp == \left[\begin{array}{l} \Xi P\_StPar\, ;\, \Xi Q_1\_StPar\, ;\, \cdots\, ;\, \Xi Q_n\_StPar; \\ l_b? : T_{l_b} \mid \exists\, u' : T_u \bullet post[l_b?/l_b] \end{array}\right]\right)\right) \\ \\ \left(\begin{array}{l} channel\ P\_coerOp : \Phi\left(T_{l_b}\right) \\ HIDE\_CSPB = \{\!| P\_coerOp |\!\} \\ P\_coerOp!l_b \rightarrow SKIP \end{array}\right) \end{cases}$$

$$[\Omega_1 \text{ Rule 1 and } \Phi \text{ Rule } 21]$$

$$= \begin{cases} \Omega_3\left(\ P\_coerOp \mathrel{\widehat=} \left[\begin{array}{l} \Xi P\_StPar\, ;\, \Xi Q_1\_StPar\, ;\, \cdots\, ;\, \Xi Q_n\_StPar; \\ l_b? : T_{l_b} \mid \exists\, u' : T_u \bullet post[l_b?/l_b] \end{array}\right]\right) \\ \\ \left(\begin{array}{l} channel\ P\_coerOp : \Phi\left(T_{l_b}\right) \\ HIDE\_CSPB = \{\!| P\_coerOp |\!\} \\ P\_coerOp!l_b \rightarrow SKIP \end{array}\right) \end{cases}$$

$$[\Omega_2 \text{ Rule } 4]$$

□

E.8.3.7 Parametrisation By Value

Link Rule 58 (Parametrisation By Value).

$$\Upsilon\left(\left(\mathbf{val}\, x : T \bullet A\right)(e)\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\left(\mathbf{val}\, x : T \bullet A\right)(e)\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left(\left(\mathbf{val}\, x : T \bullet A\right)(e)\right)\right) & \text{Behaviour Part} \end{cases} \qquad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\mathbf{var}\, x : T \bullet \left(x := e\, ;\, A\right)\right)\right) & \text{State Part} \\ \\ \Phi\left(R_{wrt}\left(\mathbf{var}\, x : T \bullet \left(x := e\, ;\, A\right)\right)\right) & \text{Behaviour Part} \end{cases} \qquad [R_{wt} \text{ Rule } 42]$$

$$= \Phi\left(R_{pre}\left(x := e\, ;\, A\right)\right) \rightarrow \textstyle\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\, ;\, A\right)\right), \{x\}\right)$$

$$\text{[Link Rule 54]}$$

$$= \Phi\left(R_{pre}\left(x := e\right)\right) \rightarrow \textstyle\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\right)\, ;\, R_{wrt}\left(A\right)\right), \{x\}\right)$$

$$[R_{wrt} \text{ Rule } 29]$$

$$= \textstyle\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\right)\, ;\, R_{wrt}\left(A\right)\right), \{x\}\right) \qquad [R_{wrt} \text{ Rule } 39]$$

$$= \textstyle\bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\right)\right)\, ;\, \Phi\left(R_{wrt}\left(A\right)\right), \{x\}\right) \qquad [\Phi \text{ Rule } 26]$$

$$= \begin{cases} \Omega_3\left(\ P\_assOp \mathrel{\widehat{=}} \cdots\ \right) & \text{State Part} \\ \left(\begin{array}{l} channel\ P\_assOp : \cdots \\ HIDE\_CSPB = \{\!|P\_assOp|\!\} \\ \sqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\begin{array}{l}\left(\begin{array}{l}(P\_assOp! \cdots \to SKIP)\,; \\ \Phi\left(R_{wrt}\left(A\right)\right)\end{array}\right) \\ , \{x\}\end{array}\right)\end{array}\right) & \text{Behaviour Part} \end{cases}$$

[Link Rule 52]

□

E.8.3.8 Parametrisation By Result

Link Rule 59 (Parametrisation By Result).

$$\Upsilon\left(\left(\mathbf{res}\, x : T \bullet A\right)(y)\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\left(\mathbf{res}\, x : T \bullet A\right)(y)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\left(\mathbf{res}\, x : T \bullet A\right)(y)\right)\right) & \text{Behaviour Part} \end{cases} \quad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\mathbf{var}\, x : T \bullet \left(A\,;\, y := x\right)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\mathbf{var}\, x : T \bullet \left(A\,;\, y := x\right)\right)\right) & \text{Behaviour Part} \end{cases} \quad [R_{wt} \text{ Rule 42}]$$

$$= \Phi\left(R_{pre}\left(A\,;\, y := x\right)\right) \to \sqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(A\,;\, y := x\right)\right), \{x\}\right)$$

[Link Rule 54]

$$= \Phi\left(R_{pre}\left(A\right)\right) \to \sqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(A\right)\,;\, R_{wrt}\left(y := x\right)\right), \{x\}\right)$$

[R_{wrt} Rule 29]

$$= \Phi\left(R_{pre}\left(A\right)\right) \to \sqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(A\right)\right)\,;\, \Phi\left(R_{wrt}\left(y := x\right)\right), \{x\}\right)$$

[Φ Rule 26]

$$= \begin{cases} \Omega_3\left(\ P\_assOp \mathrel{\widehat{=}} \cdots\ \right) & \text{State Part} \\ \left(\begin{array}{l} channel\ P\_assOp : \cdots \\ HIDE\_CSPB = \{\!|P\_assOp|\!\} \\ \Phi\left(R_{pre}\left(A\right)\right) \to \sqcap_{x:\Phi(T)} \bullet \\ \qquad F_{Mem}\left(\left(\begin{array}{l}\Phi\left(R_{post}\left(A\right)\right); \\ (P\_assOp? \cdots \to SKIP)\end{array}\right), \{x\}\right)\end{array}\right) & \text{Behaviour Part} \end{cases}$$

[Link Rule 52]

□

E.8.3.9 Parametrisation By Value-Result

Link Rule 60 (Parametrisation By Value-Result).

$$\Upsilon\left(\left(\mathbf{vres}\, x : T \bullet A\right)(y)\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\left(\mathbf{vres}\, x : T \bullet A\right)(y)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\left(\mathbf{vres}\, x : T \bullet A\right)(y)\right)\right) & \text{Behaviour Part} \end{cases} \quad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(\mathbf{var}\, x : T \bullet \left(x := y\,;\, A\,;\, y := x\right)\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(\mathbf{var}\, x : T \bullet \left(x := y\,;\, A\,;\, y := x\right)\right)\right) & \text{Behaviour Part} \end{cases} \quad [R_{wt} \text{ Rule 42}]$$

$$= \left(\begin{array}{l} \Phi\left(R_{pre}\left(x := y \,;\, A \,;\, y := x\right)\right) \rightarrow \\ \bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := y \,;\, A \,;\, y := x\right)\right), \{x\}\right) \end{array}\right) \quad \text{[Link Rule 54]}$$

$$= \left(\begin{array}{l} \Phi\left(R_{pre}\left(x := y\right)\right) \rightarrow \\ \bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\right) ; R_{wrt}\left(A\right) ; R_{wrt}\left(y := x\right)\right), \{x\}\right) \end{array}\right)$$

$$\text{[}R_{wrt}\text{ Rule 29]}$$

$$= \bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\right) ; R_{wrt}\left(A\right) ; R_{wrt}\left(y := x\right)\right), \{x\}\right)$$

$$\text{[}R_{wrt}\text{ Rule 39]}$$

$$= \bigsqcap_{x:\Phi(T)} \bullet F_{Mem}\left(\Phi\left(R_{post}\left(x := e\right)\right) ; \Phi\left(R_{wrt}\left(A\right)\right) ; \Phi\left(R_{wrt}\left(y := x\right)\right), \{x\}\right)$$

$$\text{[}\Phi\text{ Rule 26]}$$

$$= \begin{cases} \Omega_3\left(\begin{array}{l} P\_assOp1 \mathrel{\widehat=} \cdots \\ P\_assOp2 \mathrel{\widehat=} \cdots \end{array}\right) & \text{State Part} \\ \left(\begin{array}{l} channel\ P\_assOp1 : \cdots \\ channel\ P\_assOp2 : \cdots \\ HIDE\_CSPB = \{\!|P\_assOp1, P\_assOp2|\!\} \\ \bigsqcap_{x:\Phi(T)} \bullet \\ \qquad F_{Mem}\left(\begin{array}{l}\left(\begin{array}{l} (P\_assOp1! \cdots \rightarrow SKIP) ; \\ \Phi\left(R_{wrt}\left(A\right)\right) ; \\ (P\_assOp2! \cdots \rightarrow SKIP) \end{array}\right) \\ , \{x\} \end{array}\right) \end{array}\right) & \text{Behaviour Part} \end{cases}$$

$$\text{[Link Rule 52]}$$

□

E.8.4 Renaming

Link Rule 61 (Renaming).

$$\Upsilon\left(A[v_{old} := v_{new}]\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(A[v_{old} := v_{new}]\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(A[v_{old} := v_{new}]\right)\right) & \text{Behaviour Part} \end{cases} \quad \text{[Link Definition]}$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(A[v_{new}/v_{old}]\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(A[v_{new}/v_{old}]\right)\right) & \text{Behaviour Part} \end{cases} \quad \text{[}R_{wt}\text{ Rule 46]}$$

Then after that it is linked by other defined Link Rules in Section E.8. □

E.8.5 Action Invocation

Link Rule 62 (Action Invocation (without implicit recursion)). If an action A does not occur in its body of the definition, $B(A)$, then

$$\Upsilon\left(A\right)$$

$$= \begin{cases} \Omega\left(R_{wrt}\left(A\right)\right) & \text{State Part} \\ \Phi\left(R_{wrt}\left(A\right)\right) & \text{Behaviour Part} \end{cases} \quad \text{[Link Definition]}$$

$$= \Phi\left(R_{wrt}\left(A\right)\right) \quad \text{[Only behavioural part in this construct]}$$

$$= \Phi\left(R_{wrt}\left(B(A)\right)\right) \quad \text{[}R_{wrt}\text{ Rule 35]}$$

Finally, the body of the action A is linked by other Link Rules in Section E.8. □

Link Rule 63 (Action Invocation (with implicit recursion)). If an action A occurs in its body of the definition, $B(A)$, then

$$\begin{aligned}
&\Upsilon(A) \\
&= \begin{cases} \Omega(R_{wrt}(A)) & \text{State Part} \\ \Phi(R_{wrt}(A)) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \Phi(R_{wrt}(A)) && \text{[Only behavioural part in this construct]} \\
&= \Phi(R_{wrt}(\mu A \bullet B(A))) && [R_{wrt} \text{ Rule 35}] \\
&= \Phi(\mu A \bullet R_{wrt}(B(A))) && [R_{wrt} \text{ Rule 34}] \\
&= \text{let X} = \Phi(R_{wrt}(B(A))) \text{ within X} && [\Phi \text{ Rule 34}]
\end{aligned}$$

Finally, the body of the action A is linked by other Link Rules in Section E.8. □

E.8.6 Parametrised Action

E.8.6.1 Unnamed Parametrised Action Invocation

Link Rule 64 (Unnamed Parametrised Action Invocation).

$$\begin{aligned}
&\Upsilon((x : T \bullet A)(e)) \\
&= \begin{cases} \Omega(R_{wrt}((x : T \bullet A)(e))) & \text{State Part} \\ \Phi(R_{wrt}((x : T \bullet A)(e))) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \Phi(R_{wrt}((x : T \bullet A)(e))) && \text{[Only behavioural part in this construct]} \\
&= \Phi(R_{wrt}(A[e/x])) && [R_{wrt} \text{ Rule 36}]
\end{aligned}$$

Finally, the link of an unnamed parametrised action invocation is equal to the link of the unnamed parametrised action's body, but all formal parameters are substituted by corresponding expressions e, by other Link Rules in Section E.8. □

E.8.6.2 Parametrised Action Invocation

Link Rule 65 (Parametrised Action Invocation). Provided A is a parametrised action defined below,

$$PA \mathrel{\widehat=} x : T \bullet A$$

then

$$\begin{aligned}
&\Upsilon(PA(e)) \\
&= \begin{cases} \Omega(R_{wrt}(PA(e))) & \text{State Part} \\ \Phi(R_{wrt}(PA(e))) & \text{Behaviour Part} \end{cases} && \text{[Link Definition]} \\
&= \Phi(R_{wrt}(PA(e))) && \text{[Only behavioural part in this construct]} \\
&= \Phi(R_{wrt}(A[e/x])) && [R_{wrt} \text{ Rule 37}]
\end{aligned}$$

Finally, the link of a parametrised action invocation is equal to the link of the parametrised action's body, but all formal parameters are substituted by corresponding expressions e, by other Link Rules in Section E.8. □

Appendix F

Translator

The source code of the translator is available either in accompanying CD-ROM or online at GitHub [105].

Appendix G

Reactive Buffer and linked Models

The *Circus* models and the corresponding resultant $CSP \parallel_B Z$ models for both specification and implementation are available either in accompanying CD-ROM or online at GitHub [106]. In this appendix, only *Circus* models are illustrated below.

G.1 Specification

G.1.1 *Circus* Model

section $BufferSpec$ **parents** $circus\_toolkit$

$$maxbuff : \mathbb{N}_1$$

channel $input, output : \mathbb{N}$

process $Buffer \mathrel{\widehat=}$ **begin**

$$
\begin{array}{l}
\quad \textbf{state}\ BufferState == [\, buff : \operatorname{seq} \mathbb{N} ; size : 0 \mathrel{..} maxbuff \mid \\
\qquad size = \# buff \leq maxbuff \,] \\
\quad BufferInit == [\, (BufferState)' \mid buff' = \langle\rangle \land size' = 0 \,] \\
\quad InputCmd == [\, \Delta BufferState ; x? : \mathbb{N} \mid size < maxbuff \land \\
\qquad buff' = buff \frown \langle x? \rangle \land size' = size + 1 \,] \\
\quad Input \mathrel{\widehat=} (size < maxbuff) \mathbin{\&} input?x \rightarrow \big(InputCmd\big) \\
\quad OutputCmd == [\, \Delta BufferState \mid size > 0 \land buff' = tail\ buff \land \\
\qquad size' = size - 1 \,] \\
\quad Output \mathrel{\widehat=} (size > 0) \mathbin{\&} output!(head\ buff) \rightarrow \big(OutputCmd\big) \\
\quad \bullet \big(BufferInit\big) ; (\mu X \bullet (Input \mathrel{\Box} Output) ; X)
\end{array}
$$

end

G.1.2 Resultant CSP and Z Model

Available either in accompanying CD-ROM or online at GitHub [106].

G.2 Implementation

G.2.1 *Circus* Model

section $DisBufferSpec$ **parents** $circus\_toolkit$

$$\begin{array}{|l} maxbuff : \mathbb{N}_1 \\ maxring : \mathbb{N}_1 \\ \hline maxring = maxbuff - 1 \end{array}$$

$$RingIndex \;==\; 1 \,..\, maxring$$

$$\begin{array}{l}
\mathbf{channel}\ input, output : \mathbb{N} \\
\mathbf{channel}\ read, write : (RingIndex) \times \mathbb{N} \\
\mathbf{channel}\ rd, wrt : \mathbb{N} \\
\mathbf{channel}\ rd\_i, wrt\_i : (RingIndex) \times \mathbb{N}
\end{array}$$

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

$$\begin{array}{l}
\mathbf{state}\ ControllerState == [\\
\quad size : 0 \,..\, maxbuff; \\
\quad ringsize : 0 \,..\, maxring; \\
\quad cache : \mathbb{N}; \\
\quad top, bot : RingIndex \mid \\
\quad ringsize \bmod maxring = (top - bot) \bmod maxring \land \\
\quad ringsize = max \{ 0, size - 1 \}] \\
ControllerInit == [(ControllerState)' \mid \\
\quad top' = 1 \land bot' = 1 \land size' = 0]
\end{array}$$

$$\begin{array}{l}
CacheInput == [\Delta ControllerState ; x? : \mathbb{N} \mid \\
\quad size = 0 \land size' = 1 \land \\
\quad cache' = x? \land bot' = bot \land top' = top]
\end{array}$$

$$\begin{array}{l}
StoreInputController == [\Delta ControllerState \mid \\
\quad 0 < size \land size < maxbuff \land size' = size + 1 \land \\
\quad cache' = cache \land bot' = bot \land top' = (top \bmod maxring) + 1]
\end{array}$$

$$\begin{array}{l}
InputController \mathrel{\widehat=} (size < maxbuff) \mathbin{\&} input?x \rightarrow \\
\quad ((size = 0) \mathbin{\&} \big(CacheInput\big) \;\Box \\
\quad (size > 0) \mathbin{\&} write.top!x \rightarrow \big(StoreInputController\big))
\end{array}$$

$$\begin{array}{l}
NoNewCache == [\Delta ControllerState \mid \\
\quad size = 1 \land size' = 0 \land \\
\quad cache' = cache \land bot' = bot \land top' = top]
\end{array}$$

$$\begin{array}{l} StoreNewCacheController == [\,\Delta ControllerState\,;\, x? : \mathbb{N} \mid \\ \quad size > 1 \land size' = size - 1 \land \\ \quad cache' = x? \land bot' = (bot \bmod maxring) + 1 \land \\ \quad top' = top\,] \end{array}$$

$$\begin{array}{l} OutputController \mathrel{\widehat=} (size > 0)\ \&\ output!(cache) \rightarrow \\ \quad ((size > 1)\ \&\ read.bot?x \rightarrow \big(StoreNewCacheController\big) \mathrel{\square} \\ \quad (size = 1)\ \&\ \big(NoNewCache\big)) \\ \bullet \big(ControllerInit\big)\,;\,(\mu X \bullet (InputController \mathrel{\square} OutputController)\,;\, X) \end{array}$$

end

$$\begin{array}{l} \textbf{process}\ \ RingCell \mathrel{\widehat=} \textbf{begin} \\ \quad \textbf{state}\ \ CellState == [\, v : \mathbb{N} \mid true\,] \\ \quad CellWrite\ \ ==\ \ [\,\Delta CellState\,;\, x? : \mathbb{N} \mid v' = x?\,] \\ \quad Read \mathrel{\widehat=} rd!v \rightarrow \textbf{Skip} \\ \quad Write \mathrel{\widehat=} wrt?x \rightarrow \big(CellWrite\big) \\ \quad \bullet\ (\mu X \bullet (Read \mathrel{\square} Write)\,;\, X) \\ \textbf{end} \end{array}$$

$$\textbf{process}\ \ IRCell \mathrel{\widehat=} (\, i : RingIndex \odot RingCell\,)[rd\_i, wrt\_i := read, write]$$

$$\textbf{process}\ \ Ring \mathrel{\widehat=} (\, \mathop{|||}\, i : RingIndex \bullet IRCell\lfloor i \rfloor\,)$$

$$\textbf{process}\ \ Buffer \mathrel{\widehat=} (\, Controller\ [\![\, \{\!|\, read, write\,|\!\}\,]\!]\ Ring\,) \setminus \{\!|\, read, write\,|\!\}$$

G.2.2 Resultant CSP and Z Model

Available either in accompanying CD-ROM or online at GitHub [106].

Appendix H

ESEL and linked Models

The *Circus* models and the corresponding resultant $CSP \parallel_B Z$ models for the Specification, the System One, and the System Two are available either in accompanying CD-ROM or online at GitHub [107]. In this appendix, only *Circus* models are illustrated below.

H.1 Header

$\mathbf{section}\ \ ESELHeader\ \mathbf{parents}\ \ circus\_toolkit$

$MAX\_ESEL : \mathbb{N}$
$MAX\_PID : \mathbb{N}$

$ESID ::= ES\langle\langle 1 \mathrel{..} MAX\_ESEL \rangle\rangle$
$PID ::= PD\langle\langle 1 \mathrel{..} MAX\_PID \rangle\rangle$

$Price == \mathbb{N}$

$UStatus ::= uok \mid ufail$

$FStatus ::= fail\langle\langle ESID \rangle\rangle \mid NA$

$\mathbf{channel}\ \ updateallmap : ESID \pfun PID$
$\mathbf{channel}\ \ updatemap : ESID \pfun PID$

$\mathbf{channel}\ \ updateallprice : PID \pfun Price$
$\mathbf{channel}\ \ updateprice : PID \pfun Price$

$\mathbf{channel}\ \ update$

$\mathbf{channel}\ \ failures : PID \pfun \mathbf{P}\ FStatus$

$\mathbf{channel}\ \ resp : PID \times FStatus$
$\mathbf{channel}\ \ terminate$
$\mathbf{channelset}\ \ RespInterface == \{\!| resp, terminate |\!\}$

$$\begin{array}{l}
\mathbf{channel}\ \ uupdate : ESID \times Price \\
\mathbf{channel}\ \ ures : ESID \times UStatus \\
\mathbf{channel}\ \ uinit, finishuinit \\
\mathbf{channel}\ \ udisplay, finishudisplay
\end{array}$$

$$\begin{array}{l}
\mathbf{channel}\ \ init, finishinit \\
\mathbf{channel}\ \ display, finishdisplay
\end{array}$$

$$\begin{array}{l}
\mathbf{channel}\ \ write : ESID \times Price \\
\mathbf{channel}\ \ read : ESID \times Price \\
\mathbf{channel}\ \ ondisplay : ESID \\
\mathbf{channel}\ \ offdisplay : ESID
\end{array}$$

H.2 Specification

H.2.1 *Circus* Model

$\mathbf{section}\ \ ESELSpec\ \mathbf{parents}\ \ ESELHeader$

$$\begin{array}{l}
\mathbf{process}\ \ Controller \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ \ State == [\, pumap : ESID \pfun PID\, ;\, ppmap : PID \pfun Price; \\
\qquad response : PID \pfun (\mathbb{P}\ \ FStatus)\,] \\
\quad Init == [\, (State)' \mid pumap' = \varnothing \land ppmap' = \varnothing \land response' = \varnothing\,] \\
\quad UpdateMap == [\, \Delta State\, ;\, map? : ESID \pfun PID \mid \\
\qquad pumap' = pumap \oplus map? \land ppmap' = ppmap \land response' = response\,] \\
\quad UpdateAllMap == [\, \Delta State\, ;\, map? : ESID \pfun PID \mid \\
\qquad pumap' = map? \land ppmap' = ppmap \land response' = response\,] \\
\quad NewPrice == [\, \Delta State\, ;\, price? : PID \pfun Price \mid \\
\qquad ppmap' = ppmap \oplus price? \land pumap' = pumap \land response' = response\,] \\
\quad AllNewPrice == [\, \Delta State\, ;\, price? : PID \pfun Price \mid \\
\qquad ppmap' = price? \land pumap' = pumap \land response' = response\,] \\
\quad AUpdatemap \mathrel{\widehat=} updatemap?map \to \big(UpdateMap\big) \\
\qquad \Box\ updateallmap?map \to \big(UpdateAllMap\big) \\
\quad ANewPrice \mathrel{\widehat=} updateprice?price \to \big(NewPrice\big) \\
\qquad \Box\ updateallprice?price \to \big(AllNewPrice\big) \\
\quad AUpdateUnitPrice \mathrel{\widehat=} uid : ESID\, ;\, pid : PID \bullet \\
\qquad write.uid.(ppmap\ pid) \to read.uid?y \to \\
\qquad ((y = (ppmap\ pid))\ \&\ \mathbf{Skip} \\
\qquad \Box\ (y \neq (ppmap\ pid))\ \&\ resp.pid.(fail\ uid) \to \mathbf{Skip}) \\
\quad AUpdateProductUnits \mathrel{\widehat=} pid : PID \bullet \\
\qquad (\interleave\ uid : (\operatorname{dom}(pumap \rhd \{pid\}))\ \|[\,\varnothing\,]\|\ \bullet AUpdateUnitPrice(uid, pid)) \\
\quad AUpdateNoUnit \mathrel{\widehat=} pid : PID \bullet resp.pid.NA \to \mathbf{Skip} \\
\quad AUpdateProduct \mathrel{\widehat=} pid : PID \bullet \\
\qquad\ \ (pid \in \operatorname{ran}\ pumap)\ \&\ AUpdateProductUnits(pid) \\
\qquad \Box\ (pid \notin \operatorname{ran}\ pumap)\ \&\ AUpdateNoUnit(pid) \\
\quad AUpdateProducts \mathrel{\widehat=} ((\interleave\ pid : (\operatorname{dom}\ ppmap)\ \|[\,\varnothing\,]\|\ \bullet AUpdateProduct(pid)) \\
\qquad ;terminate \to \mathbf{Skip})
\end{array}$$

```
        $AddOneFailure == [\,\Delta State ; pid? : PID ; fst? : FStatus \mid$
            $(pid? \in \mathrm{dom}\ response \Rightarrow$
                $response' = response \oplus \{pid? \mapsto (response(pid?) \cup \{fst?\})\}) \land$
            $(pid? \notin \mathrm{dom}\ response \Rightarrow$
                $response' = response \cup \{pid? \mapsto \{fst?\}\}) \land$
            $ppmap' = ppmap \land pumap' = pumap\,]$      $CollectResp \mathrel{\widehat=} \mu X \bullet$
            $((resp?pid?fst \to \big(AddOneFailure\big) ; X) \Box terminate \to \mathbf{Skip})$
        $AUpdateResp \mathrel{\widehat=}$
            $(AUpdateProducts \llbracket \varnothing \mid RespInterface \mid \{response\} \rrbracket CollectResp)$
            $\setminus RespInterface$


        $ADisplay \mathrel{\widehat=}$
            $(\llbracket \{\!| display |\!\} \rrbracket\ uid : ESID \bullet \llbracket \varnothing \rrbracket\ display \to ($
                $\mathbf{if}\ \ uid \notin \mathrm{dom}\ pumap \longrightarrow offdisplay.uid \to \mathbf{Skip}$
                $[\!]\ uid \in \mathrm{dom}\ pumap \longrightarrow$
                    $\mathbf{if}\ \ pumap(uid) \notin \mathrm{dom}\ ppmap \longrightarrow offdisplay.uid \to \mathbf{Skip}$
                    $[\!]\ pumap(uid) \in \mathrm{dom}\ ppmap \longrightarrow$
                        $\mathbf{if}\ \ pumap(uid) \notin \mathrm{dom}\ response \longrightarrow$
                            $ondisplay.uid \to \mathbf{Skip}$
                        $[\!]\ pumap(uid) \in \mathrm{dom}\ response \longrightarrow$
                            $\mathbf{if}\ \ (fail\ uid) \notin response(pumap(uid)) \longrightarrow$
                                $ondisplay.uid \to \mathbf{Skip}$
                            $[\!]\ (fail\ uid) \in response(pumap(uid)) \longrightarrow$
                                $offdisplay.uid \to \mathbf{Skip}$
                            $\mathbf{fi}$
                        $\mathbf{fi}$
                    $\mathbf{fi}$
                $\mathbf{fi}$
            $)) \setminus \{\!| display |\!\}$
        $AUpdatePrice \mathrel{\widehat=} update \to response := \varnothing;$
            $AUpdateResp ; ADisplay ; failures.response \to \mathbf{Skip}$
        $AInit \mathrel{\widehat=} \big(Init\big) ; (\,|||\ u : ESID \parallel\![\,\varnothing\,]\!\parallel \bullet offdisplay.u \to \mathbf{Skip})$
        $\bullet\ AInit ; (\mu X \bullet (AUpdatemap \Box ANewPrice \Box AUpdatePrice) ; X)$
    $\mathbf{end}$


    $\mathbf{process}\ \ ESELSpec \mathrel{\widehat=} Controller$
```

H.2.2 Resultant CSP and Z Model

Available either in accompanying CD-ROM or online at GitHub [107].

H.3 System One

H.3.1 *Circus* Model

$\mathbf{section}\ \ ESELSystem1\ \mathbf{parents}\ \ ESELHeader$

$\mathbf{process}\ \ Controller1 \mathrel{\widehat=} \mathbf{begin}$

$$\mathbf{state}\ State == [\, pumap : ESID \pfun PID \,;\, ppmap : PID \pfun Price; \\ response : PID \pfun (\mathbb{P}\ FStatus) \,]$$

$$Init == [\, (State)' \mid pumap' = \varnothing \land ppmap' = \varnothing \land response' = \varnothing \,]$$

$$UpdateMap == [\, \Delta State \,;\, map? : ESID \pfun PID \mid \\ pumap' = pumap \oplus map? \land ppmap' = ppmap \land response' = response \,]$$
$$UpdateAllMap == [\, \Delta State \,;\, map? : ESID \pfun PID \mid \\ pumap' = map? \land ppmap' = ppmap \land response' = response \,]$$

$$NewPrice == [\, \Delta State \,;\, price? : PID \pfun Price \mid \\ ppmap' = ppmap \oplus price? \land pumap' = pumap \land response' = response \,]$$
$$AllNewPrice == [\, \Delta State \,;\, price? : PID \pfun Price \mid \\ ppmap' = price? \land pumap' = pumap \land response' = response \,]$$

$$AUpdatemap \mathrel{\widehat=} updatemap?map \rightarrow \left(UpdateMap\right) \\ \Box\ updateallmap?map \rightarrow \left(UpdateAllMap\right)$$

$$ANewPrice \mathrel{\widehat=} updateprice?price \rightarrow \left(NewPrice\right) \\ \Box\ updateallprice?price \rightarrow \left(AllNewPrice\right)$$

$$AUpdateUnitPrice \mathrel{\widehat=} uid : ESID \,;\, pid : PID \bullet \\ uupdate.uid.(ppmap\ pid) \rightarrow ures.uid?rst \rightarrow \\ ((rst = ufail)\ \&\ resp.pid.(fail\ uid) \rightarrow \mathbf{Skip} \\ \Box\ (rst = uok)\ \&\ \mathbf{Skip})$$

$$AUpdateProductUnits \mathrel{\widehat=} pid : PID \bullet \\ (\left|\left|\left|\right.\right.\right.\ uid : (\mathrm{dom}\,(pumap \rhd \{pid\}))\ \llbracket\, \varnothing \rrbracket \bullet AUpdateUnitPrice(uid, pid))$$

$$AUpdateNoUnit \mathrel{\widehat=} pid : PID \bullet resp.pid.NA \rightarrow \mathbf{Skip}$$

$$AUpdateProduct \mathrel{\widehat=} pid : PID \bullet \\ (pid \in \mathrm{ran}\ pumap)\ \&\ AUpdateProductUnits(pid) \\ \Box\ (pid \notin \mathrm{ran}\ pumap)\ \&\ AUpdateNoUnit(pid)$$

$$AUpdateProducts \mathrel{\widehat=} ((\left|\left|\left|\right.\right.\right.\ pid : (\mathrm{dom}\ ppmap)\ \llbracket\, \varnothing \rrbracket \bullet AUpdateProduct(pid)) \\ ;terminate \rightarrow \mathbf{Skip})$$

$$\begin{array}{l}
AddOneFailure == [\,\Delta State\,;\, pid? : PID\,;\, fst? : FStatus \mid \\
\quad (pid? \in \operatorname{dom} response \Rightarrow \\
\qquad response' = response \oplus \{pid? \mapsto (response(pid?) \cup \{fst?\})\}) \land \\
\quad (pid? \notin \operatorname{dom} response \Rightarrow \\
\qquad response' = response \cup \{pid? \mapsto \{fst?\}\}) \land \\
\quad ppmap' = ppmap \land pumap' = pumap\,]
\end{array}$$

$$\begin{array}{l}
CollectResp \mathrel{\widehat=} \mu X \bullet \\
\quad ((resp?pid?fst \rightarrow \big(AddOneFailure\big)\,;\, X) \mathrel{\Box} terminate \rightarrow \mathbf{Skip})
\end{array}$$

$$\begin{array}{l}
AUpdateResp \mathrel{\widehat=} \\
\quad (AUpdateProducts \llbracket \varnothing \mid RespInterface \mid \{response\} \rrbracket\, CollectResp) \\
\quad \setminus RespInterface
\end{array}$$

$$\begin{array}{l}
AUpdatePrice \mathrel{\widehat=} update \rightarrow response := \varnothing; \\
\quad AUpdateResp\,;\, display \rightarrow finishdisplay \rightarrow failures.response \rightarrow \mathbf{Skip}
\end{array}$$

$$\begin{array}{l}
\bullet \big(Init\big)\,;\, init \rightarrow \mathbf{Skip}; \\
\quad (\mu X \bullet (AUpdatemap \mathrel{\Box} ANewPrice \mathrel{\Box} AUpdatePrice)\,;\, X)
\end{array}$$

end

$$\begin{array}{l}
\mathbf{process}\ ESEL1 \mathrel{\widehat=} eid : ESID \bullet \mathbf{begin} \\
\quad \mathbf{state}\ State == [\,price : Price\,;\, status : UStatus\,] \\
\quad Init == [\,(State)' \mid price' = 0 \land status' = uok\,] \\
\quad Update \mathrel{\widehat=} uupdate.eid?x \rightarrow price := x\,;\, write.eid.price \rightarrow read.eid?y \\
\qquad \rightarrow ((y = price) \mathrel{\&} ures.eid.uok \rightarrow status := uok \\
\qquad\quad \Box\ (y \neq price) \mathrel{\&} ures.eid.ufail \rightarrow status := ufail) \\
\quad Display \mathrel{\widehat=} display \rightarrow (\\
\qquad\quad (status = uok) \mathrel{\&} ondisplay.eid \rightarrow \mathbf{Skip} \\
\qquad \Box\ (status = ufail) \mathrel{\&} offdisplay.eid \rightarrow \mathbf{Skip}) \\
\qquad ; finishdisplay \rightarrow \mathbf{Skip} \\
\quad NotUpdateDisplay \mathrel{\widehat=} display \rightarrow offdisplay.eid \rightarrow finishdisplay \rightarrow \mathbf{Skip} \\
\quad AInit \mathrel{\widehat=} \big(Init\big)\,;\, offdisplay.eid \rightarrow init \rightarrow \mathbf{Skip} \\
\quad \bullet\ AInit\,;\, (\mu X \bullet ((Update\,;\, Display) \mathrel{\Box} NotUpdateDisplay)\,;\, X) \\
\mathbf{end}
\end{array}$$

$$\begin{array}{l}
\mathbf{channelset}\ InterESELInterface1 == \{\!|\, init, display, finishdisplay \,|\!\} \\
\mathbf{process}\ ESELS1 \mathrel{\widehat=} \big\Vert\ eid : ESID \llbracket InterESELInterface1 \rrbracket \bullet ESEL1(eid)
\end{array}$$

$$\begin{array}{l}
\mathbf{channelset}\ ESELInterface1 == \{\!|\, uupdate, ures, init, display, finishdisplay \,|\!\} \\
\mathbf{process}\ ESELSystem1 \mathrel{\widehat=} \\
\quad (Controller1 \llbracket ESELInterface1 \rrbracket\, ESELS1) \setminus ESELInterface1
\end{array}$$

H.3.2 Resultant CSP and Z Model

Available either in accompanying CD-ROM or online at GitHub [107].

H.4 System Two

H.4.1 *Circus* Model

$\mathbf{section}\ \mathit{ESELSystem2}\ \mathbf{parents}\ \mathit{ESELHeader}$

$\mathit{MAX\_GATEWAY} : \mathbb{N}$

$\mathit{GID} ::= \mathit{GW}\langle\!\langle 1 \mathrel{..} \mathit{MAX\_GATEWAY} \rangle\!\rangle$

$\mathit{gwmap} : \mathit{ESID} \rightarrow \mathit{GID}$

$\mathit{gwmap} = \{(\mathit{ES}\,1, \mathit{GW}\,1), (\mathit{ES}\,2, \mathit{GW}\,1), (\mathit{ES}\,3, \mathit{GW}\,2)\}$

$\mathbf{channel}\ \mathit{gupdateprice} : \mathit{GID} \times (\mathit{ESID} \pfun \mathit{Price})$
$\mathbf{channel}\ \mathit{gfailure} : \mathit{GID} \times \mathbf{P}\ \mathit{ESID}$

$\mathbf{channel}\ \mathit{gresp} : \mathit{ESID}$
$\mathbf{channel}\ \mathit{gterminate}$
$\mathbf{channelset}\ \mathit{GRespInterface} == \{\!|\ \mathit{gresp}, \mathit{gterminate}\ |\!\}$

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

$\mathbf{state}\ \mathit{State} == [\, \mathit{pumap} : \mathit{ESID} \pfun \mathit{PID}\,;\ \mathit{ppmap} : \mathit{PID} \pfun \mathit{Price};$
$\quad \mathit{response} : \mathit{PID} \pfun (\mathbf{P}\ \mathit{FStatus})\,]$

$\mathit{Init} == [\,(\mathit{State})' \mid \mathit{pumap}' = \varnothing \land \mathit{ppmap}' = \varnothing \land$
$\mathit{response}' = \varnothing\,]$
$\mathit{UpdateMap} == [\,\Delta \mathit{State}\,;\ \mathit{map?} : \mathit{ESID} \pfun \mathit{PID} \mid$
$\quad \mathit{pumap}' = \mathit{pumap} \oplus \mathit{map?} \land \mathit{ppmap}' = \mathit{ppmap} \land \mathit{response}' = \mathit{response}\,]$
$\mathit{UpdateAllMap} == [\,\Delta \mathit{State}\,;\ \mathit{map?} : \mathit{ESID} \pfun \mathit{PID} \mid$
$\quad \mathit{pumap}' = \mathit{map?} \land \mathit{ppmap}' = \mathit{ppmap} \land \mathit{response}' = \mathit{response}\,]$
$\mathit{NewPrice} == [\,\Delta \mathit{State}\,;\ \mathit{price?} : \mathit{PID} \pfun \mathit{Price} \mid$
$\quad \mathit{ppmap}' = \mathit{ppmap} \oplus \mathit{price?} \land \mathit{pumap}' = \mathit{pumap} \land \mathit{response}' = \mathit{response}\,]$
$\mathit{AllNewPrice} == [\,\Delta \mathit{State}\,;\ \mathit{price?} : \mathit{PID} \pfun \mathit{Price} \mid$
$\quad \mathit{ppmap}' = \mathit{price?} \land \mathit{pumap}' = \mathit{pumap} \land \mathit{response}' = \mathit{response}\,]$

$\mathit{AUpdatemap} \mathrel{\widehat=} \mathit{updatemap?map} \rightarrow \big(\mathit{UpdateMap}\big)$
$\quad \Box\ \mathit{updateallmap?map} \rightarrow \big(\mathit{UpdateAllMap}\big)$
$\mathit{ANewPrice} \mathrel{\widehat=} \mathit{updateprice?price} \rightarrow \big(\mathit{NewPrice}\big)$
$\quad \Box\ \mathit{updateallprice?price} \rightarrow \big(\mathit{AllNewPrice}\big)$
$\mathit{AUpdateUnitFail} \mathrel{\widehat=} \mathit{eid} : \mathit{ESID} \bullet \mathit{resp}.(\mathit{pumap}(\mathit{eid})).(\mathit{fail}\ \mathit{eid}) \rightarrow \mathbf{Skip}$
$\mathit{AUpdateNoUnit} \mathrel{\widehat=} \mathit{pid} : \mathit{PID} \bullet \mathit{resp}.\mathit{pid}.\mathit{NA} \rightarrow \mathbf{Skip}$

$$\begin{array}{l}
ARespNoUnit \mathrel{\widehat{=}} \interleave\, pid : (\operatorname{dom} ppmap \setminus \operatorname{ran} pumap) \parallel\!\![\, \varnothing \,]\!\!\parallel \bullet \\
\quad AUpdateNoUnit(pid) \\
AUpdateGateway \mathrel{\widehat{=}} gid : GID \bullet \\
\quad gupdateprice.gid!((\operatorname{dom}(gwmap \rhd \{gid\})) \lhd (pumap \fatsemi ppmap)) \to \\
\quad gfailure.gid?uids \to (\interleave\, uid : uids \parallel\!\![\, \varnothing \,]\!\!\parallel \bullet AUpdateUnitFail(uid))
\end{array}$$

$$\begin{array}{l}
AUpdateGateways \mathrel{\widehat{=}} \interleave\, gid : GID \parallel\!\![\, \varnothing \,]\!\!\parallel \bullet AUpdateGateway(gid) \\
AUpdateProducts \mathrel{\widehat{=}} (AUpdateGateways \parallel\!\![\, \varnothing \mid \varnothing \,]\!\!\parallel ARespNoUnit); \\
\quad terminate \to \mathbf{Skip}
\end{array}$$

$$\begin{array}{l}
AddOneFailure == [\, \Delta State \,;\, pid? : PID \,;\, fst? : FStatus \mid \\
\quad (pid? \in \operatorname{dom} response \Rightarrow \\
\qquad response' = response \oplus \{pid? \mapsto (response(pid?) \cup \{fst?\})\}) \land \\
\quad (pid? \notin \operatorname{dom} response \Rightarrow \\
\qquad response' = response \cup \{pid? \mapsto \{fst?\}\}) \land \\
\quad ppmap' = ppmap \land pumap' = pumap \,]
\end{array}$$

$$\begin{array}{l}
ACollectResp \mathrel{\widehat{=}} \mu X \bullet \\
\quad ((resp?pid?fst \to \big(AddOneFailure\big) \,;\, X) \Box\, terminate \to \mathbf{Skip}) \\
AUpdateResp \mathrel{\widehat{=}} \\
\quad (AUpdateProducts \llbracket\, \varnothing \mid RespInterface \mid \{response\} \,\rrbracket\, ACollectResp) \\
\quad \setminus RespInterface \\
AUpdatePrice \mathrel{\widehat{=}} update \to response := \varnothing; \\
\quad AUpdateResp \,;\, display \to finishdisplay \to failures.response \to \mathbf{Skip}
\end{array}$$

$$\begin{array}{l}
\bullet \big(Init\big) \,;\, init \to finishinit \to \mathbf{Skip}; \\
\quad (\mu X \bullet (AUpdatemap \,\Box\, ANewPrice \,\Box\, AUpdatePrice) \,;\, X)
\end{array}$$

end

$$
\begin{array}{l}
\textbf{process}\ \ Gateway \mathrel{\widehat{=}} gid : GID \bullet \textbf{begin} \\
\quad \textbf{state}\ \ State == [\, pumap : ESID \mathrel{+\!\!\!\!\rightarrow} Price \,;\, failed : \mathbb{P}\ ESID \,] \\
\quad Init == [\, (State)' \mid pumap' = \varnothing \land failed' = \varnothing \,] \\
\quad UpdateAllMap == [\, \Delta State \,;\, map? : ESID \mathrel{+\!\!\!\!\rightarrow} Price \mid \\
\qquad pumap' = map? \land failed' = failed \,] \\
\quad AUpdateallmap \mathrel{\widehat{=}} gupdateprice.gid?map \rightarrow \big(UpdateAllMap \big) \\
\quad AUpdateUnitPrice \mathrel{\widehat{=}} uid : ESID \bullet \\
\qquad uupdate.uid.(pumap\ uid) \rightarrow ures.uid?rst \rightarrow \\
\qquad ((rst = ufail)\ \&\ gresp!uid \rightarrow \textbf{Skip} \\
\qquad \Box\ (rst = uok)\ \&\ \textbf{Skip}) \\
\quad AUpdateAllUnits \mathrel{\widehat{=}} ((|||\ eid : (\mathrm{dom}\ pumap) \parallel\!\![\, \varnothing \,]\!\!\parallel \bullet AUpdateUnitPrice(eid)) \\
\qquad ;gterminate \rightarrow \textbf{Skip}) \\
\quad AGCollectResp \mathrel{\widehat{=}} \mu X \bullet \\
\qquad ((gresp?uid \rightarrow failed := failed \cup \{uid\} \,;\, X)\ \Box\ gterminate \rightarrow \textbf{Skip}) \\
\quad AGUpdateResp \mathrel{\widehat{=}} \\
\qquad (AUpdateAllUnits \llbracket\, \varnothing \mid GRespInterface \mid \{failed\} \,\rrbracket\ AGCollectResp) \\
\qquad \setminus GRespInterface \\
\quad AGUpdatePrice \mathrel{\widehat{=}} AUpdateallmap \,;\, failed := \varnothing; \\
\qquad AGUpdateResp \,;\, gfailure.gid!failed \rightarrow display \rightarrow udisplay \rightarrow \\
\qquad finishudisplay \rightarrow finishdisplay \rightarrow \textbf{Skip} \\
\quad \bullet \big(Init \big) \,;\, init \rightarrow uinit \rightarrow finishuinit \rightarrow finishinit \rightarrow \textbf{Skip}; \\
\qquad (\mu X \bullet (AGUpdatePrice) \,;\, X) \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textbf{process}\ \ ESEL2 \mathrel{\widehat{=}} eid : ESID \bullet \textbf{begin} \\
\quad \textbf{state}\ \ State == [\, price : Price \,;\, status : UStatus \,] \\
\quad Init == [\, (State)' \mid price' = 0 \land status' = uok \,] \\
\quad Update \mathrel{\widehat{=}} uupdate.eid?x \rightarrow price := x \,;\, write.eid.price \rightarrow read.eid?y \\
\qquad \rightarrow ((y = price)\ \&\ ures.eid.uok \rightarrow status := uok \\
\qquad\ \ \Box\ (y \neq price)\ \&\ ures.eid.ufail \rightarrow status := ufail) \\
\quad Display \mathrel{\widehat{=}} udisplay \rightarrow (\\
\qquad\ \ (status = uok)\ \&\ ondisplay.eid \rightarrow \textbf{Skip} \\
\qquad \Box\ (status = ufail)\ \&\ offdisplay.eid \rightarrow \textbf{Skip}) \\
\qquad ;finishudisplay \rightarrow \textbf{Skip} \\
\quad NotUpdateDisplay \mathrel{\widehat{=}} udisplay \rightarrow offdisplay.eid \rightarrow finishudisplay \rightarrow \textbf{Skip} \\
\quad AInit \mathrel{\widehat{=}} \big(Init \big) \,;\, uinit \rightarrow offdisplay.eid \rightarrow finishuinit \rightarrow \textbf{Skip} \\
\quad \bullet AInit \,;\, (\mu X \bullet ((Update \,;\, Display)\ \Box\ NotUpdateDisplay) \,;\, X) \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textbf{channelset}\ \ InterESELInterface2 == \lbrace\!|\, uinit, finishuinit, \\
\qquad\qquad\qquad\qquad\qquad udisplay, finishudisplay \,|\!\rbrace \\
\textbf{process}\ \ ESELS2 \mathrel{\widehat{=}} gid : GID \bullet \\
\quad (\parallel\ eid : (\mathrm{dom}\,(gwmap \rhd \{gid\})) \llbracket\, InterESELInterface2 \rrbracket \bullet ESEL2(eid))
\end{array}
$$

$$
\begin{array}{l}
\textbf{channelset}\ \ InterGWInterface2 == \lbrace\!|\, init, finishinit, display, finishdisplay \,|\!\rbrace \\
\textbf{channelset}\ \ GWESELInterface2 == \lbrace\!|\, uinit, finishuinit, uupdate, ures, \\
\qquad\qquad\qquad\qquad\qquad udisplay, finishudisplay \,|\!\rbrace \\
\textbf{process}\ \ Gateways \mathrel{\widehat{=}} \parallel\ gid : GID \llbracket\, InterGWInterface2 \rrbracket \bullet \\
\quad (Gateway(gid) \llbracket\, GWESELInterface2 \,\rrbracket\ ESELS2(gid)) \setminus GWESELInterface2
\end{array}
$$

$$
\begin{array}{l}
\textbf{channelset}\ \ ServerGWInterface == \{\!|\ init, finishinit, gupdateprice, gfailure,\\
\qquad\qquad\qquad\qquad display, finishdisplay\ |\!\}\\
\textbf{process}\ \ ESELSystem2 \mathrel{\widehat=}\\
\quad (ESELServer \ [\![\ ServerGWInterface\ ]\!]\ Gateways) \setminus ServerGWInterface
\end{array}
$$

H.4.2 Resultant CSP and Z Model

Available either in accompanying CD-ROM or online at GitHub [107].

Appendix I

Steam Boiler and linked Models

The *Circus* model and the corresponding resultant $CSP \parallel_B Z$ model are available either in accompanying CD-ROM or online at GitHub [108]. In this appendix, only *Circus* models are illustrated below.

I.1 *Circus* Model

$$\textbf{section}\ \ SteamBoiler\ \textbf{parents}\ \ circus\_toolkit$$

$$\textbf{channel}\ \ clocktick, startcycle$$

I.1.1 Timer

In the original model, the *time* is initialised to *cyclelimit* by an assignment $time := cyclelimit$. In this model, we modify it to a schema expression $\left(InitTimer\right)$. They are semantically equal. The reason of this modification is because, with this schema, in the final resultant $CSP \parallel Z$ model, *time* is initialised in the early stage (during "initialisation" of the model) instead of in the later stage by the linked assignment in CSP. This will make the model checker easier to find the initial state.

The mod operator binds more tightly than + operator (albeit, it is not the case in mathematics), thus

$$(\,time := time + 1 \bmod cycletime\,)$$

will not get the expected result. It is corrected by adding additional brackets.

```
process Timer ≙ begin
    $cycletime == 5$
    $cyclelimit == cycletime - 1$
    $Time == 0 \mathrel{..} cyclelimit$
    state $TimeState == [\, time : Time \,]$
    $InitTimer == [\, TimeState' \mid time' = cyclelimit \,]$
    $TimeOp == [\, \Delta TimeState \mid time' \geq time \,]$
    $TCycle \mathrel{\widehat=} (\, time := (time + 1) \bmod cycletime \,);$
        $(\textbf{if}\ \ time = 0 \longrightarrow startcycle \rightarrow \textbf{Skip}\ \ \square\ \ time \neq 0 \longrightarrow \textbf{Skip}\ \ \textbf{fi});$
        $clocktick \rightarrow TCycle$
    $\bullet \left(InitTimer\right) ; TCycle$
end
```

I.1.2 Analyser

I.1.2.1 Parameters

MAX_NUM and $NUMS$ are introduced just for facilitating the animation.

$$\mid\ MAX\_NUM : \mathbb{N}$$

$$NUMS == 0 \mathrel{..} MAX\_NUM$$

$$\mid\ C, P, U\_1, U\_2, W : NUMS$$

$$\begin{array}{|l} M\_1, N\_1, N\_2, M\_2 : NUMS \\ \hline M\_1 \leq N\_1 \leq N\_2 \leq M\_2 \end{array}$$

I.1.2.2 Sensor

$$Unit[X] == [\, a\_1, a\_2 : NUMS ; st : X \mid a\_1 \leq a\_2 \,]$$

$$SState ::= sokay \mid sfailed$$

$$QSensor == Unit[SState][qa\_1 / a\_1, qa\_2 / a\_2, qst/st]$$

$$InitQSensor == [\, QSensor' \mid qa\_1' = 0 \land qa\_2' = C \land qst' = sokay \,]$$

$$VSensor == Unit[SState][va\_1 / a\_1, va\_2 / a\_2, vst/st]$$

$$InitVSensor == [\, VSensor' \mid va\_1' = 0 \land va\_2' = 0 \land vst' = sokay \,]$$

I.1.2.3 Pump

$$PState ::= popen \mid pwaiting \mid pclosed \mid pfailed$$

$Pump0$ is rewritten to give a small size set $\{0, P\}$ as pa's type to ease model checking. Since the values of pa_1 and pa_2 are implied from the pump state and not the input value from environment, it is safe to reduce the size of their type.

$$Pump0 == [\, pa\_1, pa\_2 : \{0, P\} ; pst : PState \mid pa\_1 \leq pa\_2 \,]$$

$$PumpOpen == [\, Pump0 \mid pst = popen \Rightarrow (pa\_1 = P \land pa\_2 = P) \,]$$

$$\begin{array}{l} PumpWaitingOrClosed == [\, Pump0 \mid \\ \quad (pst = pwaiting \lor pst = pclosed) \Rightarrow (pa\_1 = 0 \land pa\_2 = 0) \,] \end{array}$$

$$Pump == PumpOpen \land PumpWaitingOrClosed$$

$$InitPump == [\, PumpWaitingOrClosed' \mid pst' = pclosed \,]$$

$$PCState ::= pcflow \mid pcnoflow \mid pcfailed$$
$$PumpCtr0 == [\, Pump \,;\, pcst : PCState \,]$$

$$\begin{aligned}&POpenPCFlowOrFailed == [\, PumpCtr0 \mid \\ &\quad pst = popen \Rightarrow (pcst = pcflow \lor pcst = pcfailed)\,]\end{aligned}$$

$$\begin{aligned}&PWaitingPCNoFlowOrFailed == [\, PumpCtr0 \mid \\ &\quad pst = pwaiting \Rightarrow (pcst = pcnoflow \lor pcst = pcfailed)\,]\end{aligned}$$

$$\begin{aligned}&PClosedPCNoFlowOrFailed == [\, PumpCtr0 \mid \\ &\quad pst = pclosed \Rightarrow (pcst = pcnoflow \lor pcst = pcfailed)\,]\end{aligned}$$

$$\begin{aligned}&PFailedPCFlow == [\, PumpCtr0 \mid \\ &\quad (pst = pfailed \land pcst = pcflow) \Rightarrow (pa\_1 = P \land pa\_2 = P)\,]\end{aligned}$$

$$\begin{aligned}&PFailedPCNoFlow == [\, PumpCtr0 \mid \\ &\quad (pst = pfailed \land pcst = pcnoflow) \Rightarrow (pa\_1 = 0 \land pa\_2 = 0)\,]\end{aligned}$$

$$\begin{aligned}&PFailedPCFailed == [\, PumpCtr0 \mid \\ &\quad (pst = pfailed \land pcst = pcfailed) \Rightarrow (pa\_1 = 0 \land pa\_2 = P)\,]\end{aligned}$$

$$\begin{aligned}&PumpCtr == \\ &\quad POpenPCFlowOrFailed \land PWaitingPCNoFlowOrFailed \land \\ &\quad PClosedPCNoFlowOrFailed \land PFailedPCFlow \land PFailedPCNoFlow \land \\ &\quad PFailedPCFailed\end{aligned}$$

$$InitPumpCtr == [\, PumpCtr' \mid InitPump \land pcst' = pcnoflow \,]$$

$$PumpIndex == 1 \mathinner{\ldotp\ldotp} 4$$

The names of pa_1 and pa_2 are changed to pta_1 and pta_2 to avoid confusion. And their types are changed as well due to the same reason as pa_1 and pa_2 in $Pump0$.

PumpCtrSystem

$$\begin{aligned}&pumpctr : PumpIndex \to PumpCtr \\ &pta\_1 \,,\, pta\_2 : \{0, P, 2 * P, 3 * P, 4 * P\}\end{aligned}$$

$$\begin{aligned}&pta\_1 = (pumpctr\ 1).pa\_1 + (pumpctr\ 2).pa\_1 + \\ &\quad (pumpctr\ 3).pa\_1 + (pumpctr\ 4).pa\_1 \\[1ex] &pta\_2 = (pumpctr\ 1).pa\_2 + (pumpctr\ 2).pa\_2 + \\ &\quad (pumpctr\ 3).pa\_2 + (pumpctr\ 4).pa\_2\end{aligned}$$

InitPumpCtrSystem

$$PumpCtrSystem'$$

$$\begin{aligned}&\exists\, InitPumpCtr \bullet \\ &\quad \forall\, i : PumpIndex \bullet pumpctr'\ i = \theta\, PumpCtr'\end{aligned}$$

I.1.2.4 Valve

A freetype *VAction* and a schema *SetValveState* are added to update valve's state according to the output signal sent to the physical units. If this program sends *openValve* (or *closeValve*), then its action is *openv* (or *closev*) and its state should be *vopen* (or *vclosed*). Otherwise, if none of *openValve* and *closeValve* is issued, then it is *VNoChange* and its state is unchanged.

$$VState ::= vopen \mid vclosed$$

$$VAction ::= openv \mid closev \mid VNoChange$$

$$Valve == [\, valve : VState \,]$$

$$InitValve == [\, Valve' \mid valve' = vclosed \,]$$

$$\begin{aligned}&SetValveState == [\, \Delta Valve ; vstate? : VAction \mid \\ &\quad (vstate? = VNoChange \Rightarrow valve' = valve) \land \\ &\quad (vstate? = openv \Rightarrow valve' = vopen) \land \\ &\quad (vstate? = closev \Rightarrow valve' = vclosed) \,]\end{aligned}$$

I.1.2.5 Expected values

$$CValues == [\, qc\_1 , qc\_2 , vc\_1 , vc\_2 : NUMS \,]$$

$$InitCValues == [\, CValues' \mid qc\_1' = 0 \land qc\_2' = C \land vc\_1' = 0 \land vc\_2' = W \,]$$

$$QLowerBoundValveOpen == [\, CValues ; Valve \mid valve = vopen \land qc\_1 = 0 \,]$$

$$\begin{aligned}&QLowerBoundValveClosed == \\ &\quad [\, CValues ; QSensor ; VSensor ; PumpCtrSystem ; Valve \mid valve = vclosed \land \\ &\qquad qc\_1 = max\{0, qa\_1 - 5 * va\_2 - 12 * U\_1 + 5 * pta\_1\} \,]\end{aligned}$$

qc_2 must be larger than or equal to 0.

$$\begin{aligned}&QUpperBound == \\ &\quad [\, CValues ; QSensor ; VSensor ; PumpCtrSystem \mid \\ &\qquad qc\_2 = max\{0, min\{C, qa\_2 - 5 * va\_1 + 12 * U\_2 + 5 * pta\_2\}\} \,]\end{aligned}$$

$$VLowerBound == [\, CValues ; VSensor \mid vc\_1 = max\{0, va\_1 - 5 * U\_2\} \,]$$

$vc\_2 = min\{W, va\_2 - 5 * U\_1\}$ should be $vc\_2 = min\{W, va\_2 + 5 * U\_1\}$.

$$VUpperBound == [\, CValues ; VSensor \mid (vc\_2 = min\{W, va\_2 + 5 * U\_1\}) \,]$$

$$InputPState == \{popen, pclosed\}$$

$$InputPCState == \{pcflow, pcnoflow\}$$

$$\begin{array}{|l}\hline ExpectedPumpStates \\ expectedp : PumpIndex \rightarrow InputPState \\ expectedpc : PumpIndex \rightarrow InputPCState \\ \hline\end{array}$$

We add a schema *InitExpectedPumpStates* to initialise the expected pump states though their initial states can be arbitrarily chosen. In addition, we use abnormal combination of the pump state *pclosed* and the pump controller state *pcflow* to indicate this initial value should not be used to check again input pump and pump controller states.

```
InitExpectedPumpStates
  $ExpectedPumpStates'$
  ----------------
  $expectedp' = \{1 \mapsto pclosed, 2 \mapsto pclosed, 3 \mapsto pclosed, 4 \mapsto pclosed\}$
  $expectedpc' = \{1 \mapsto pcflow, 2 \mapsto pcflow, 3 \mapsto pcflow, 4 \mapsto pcflow\}$
```

This schema *CalcExpectedPumpState* is added to update expected pump and pump controller states according to output pump states to the physical units. If the output pump state is *popen*, then the expected pump state is *popen* as well and the pump controller state will be *pcflow*. Otherwise, *pclosed* and *pcnoflow* respectively. At the same time, the pump state is changed to *pwaiting* in case the pump is expected to be opened from closed.

```
CalcExpectedPumpState
  $\Delta ExpectedPumpStates$
  $\Delta PumpCtrSystem$
  $pumpstate? : PumpIndex \rightarrow InputPState$
  ----------------
  $\forall i : PumpIndex \bullet$
      $($
          $(expectedp'\ i = pumpstate?\ i) \land$
          $($
              $(pumpstate?\ i = popen \land expectedpc'\ i = pcflow) \lor$
              $(pumpstate?\ i = pclosed \land expectedpc'\ i = pcnoflow)$
          $)$
      $) \land$
      $($
          $((pumpctr'\ i).pst =$
              $\mathbf{if}(expectedp\ i = pclosed \land$
                  $pumpstate?\ i = popen \land$
                  $(pumpctr\ i).pst = pclosed)$
              $\mathbf{then}$
                  $pwaiting$
              $\mathbf{else}$
                  $(pumpctr\ i).pst$
          $) \land$
          $(pumpctr'\ i).pcst = (pumpctr\ i).pcst$
      $)$
```

$$\begin{aligned}&Equipment0 == \\ &\quad QSensor \land VSensor \land PumpCtrSystem \land Valve \land \\ &\quad CValues \land ExpectedPumpStates\end{aligned}$$

I.1.2.6 Failures and repairs

$$QFailed == [\, QSensor \mid qst = sfailed \,]$$

$$VFailed == [\, VSensor \mid vst = sfailed \,]$$

$$\mathit{PFailed} == [\, \mathit{PumpCtrSystem} \mid (\, \exists\, i : \mathit{PumpIndex} \bullet (\mathit{pumpctr}\ i).\mathit{pst} = \mathit{pfailed}\,)\,]$$

$$\mathit{PCFailed} == [\, \mathit{PumpCtrSystem} \mid (\, \exists\, i : \mathit{PumpIndex} \bullet (\mathit{pumpctr}\ i).\mathit{pcst} = \mathit{pcfailed}\,)\,]$$

$$\mathit{UnitFailure} ::= \mathit{qfail} \mid \mathit{vfail} \mid \mathit{pfail}\langle\!\langle \mathit{PumpIndex} \rangle\!\rangle \mid \mathit{pcfail}\langle\!\langle \mathit{PumpIndex} \rangle\!\rangle$$

$$\mathit{Failures} == [\, \mathit{failures}, \mathit{noacks} : \mathbf{P}\ \mathit{UnitFailure} \mid \mathit{noacks} \subseteq \mathit{failures}\,]$$

The original schema uses

$$(\, u = \mathit{pfail}\ i \wedge \mathit{PFailed}\,)$$

to calculate pump failures. However, since *PFailed* holds if at least one of pumps is failed, the schema results in pump failures for all pumps. Finally, the schema is updated to check pump failures against individual pump state directly by

$$(\, u = \mathit{pfail}\ i \wedge (\mathit{pumpctr}\ i).\mathit{pst} = \mathit{pfailed}\,)$$

. This is the same case as *pcfail*.

```
┌─ EquipmentFailures ─────────────────────────────
│ Equipment0
│ Failures
├──────────
│ failures =
│     { u : UnitFailure ; i : PumpIndex |
│         ( u = qfail ∧ QFailed ) ∨
│         ( u = vfail ∧ VFailed ) ∨
│         ( u = pfail i ∧ (pumpctr i).pst = pfailed ) ∨
│         ( u = pcfail i ∧ (pumpctr i).pcst = pcfailed )
│     • u }
└─────────────────────────────────────────────────
```

$$\mathit{InitFailures} == [\, \mathit{Failures}' \mid \mathit{failures}' = \varnothing \wedge \mathit{noacks}' = \varnothing\,]$$

$$\mathit{FailuresExpected} == [\, \mathit{Failures} ; \mathit{failureacks} : \mathbf{P}\ \mathit{UnitFailure} \mid \mathit{failureacks} \subseteq \mathit{noacks}\,]$$

$$\mathit{AcceptFailureAcks} == [\, \Delta \mathit{Failures} ; \mathit{FailuresExpected} \mid \mathit{noacks}' = \mathit{noacks} \setminus \mathit{failureacks}\,]$$

$$\mathit{RepairsExpected} == [\, \mathit{Failures} ; \mathit{repairs} : \mathbf{P}\ \mathit{UnitFailure} \mid \mathit{repairs} \subseteq \mathit{failures}\,]$$

$$\mathit{AcceptRepairs} == [\, \Delta \mathit{Failures} ; \mathit{RepairsExpected} \mid \mathit{failures}' = \mathit{failures} \setminus \mathit{repairs} \wedge \mathit{noacks}' = \mathit{noacks} \setminus \mathit{repairs}\,]$$

The schema *UpdateFailuresAck* is added to update *noacks* according to input *failureacks*? and *repairs*?.

- For the new failures identified in this cycle, we add them to *noacks* to state they are not acknowledged.
- If *failureacks*? is accepted, that is $\mathit{failureacks}? \subseteq \mathit{noacks}$, we take these acknowledged failures out of *noacks*.

- If *repairs*? is accepted, that is *repairs*? $\subseteq$ *failures*, we take these repaired failures out of *noacks*.

$$
\begin{array}{l}
\textit{UpdateFailuresAck} \\
\hline
\Delta \textit{Failures} \\
\textit{failureacks?} : \mathbf{P}\ \textit{UnitFailure} \\
\textit{repairs?} : \mathbf{P}\ \textit{UnitFailure} \\
\hline
\exists\, \textit{newnoacks} : \mathbf{P}\ \textit{UnitFailure} \bullet (\\
\quad (\textit{newnoacks} = \textit{noacks} \cup (\textit{failures}' \setminus \textit{failures})) \land \\
\quad (\\
\qquad (((\textit{failureacks?} \subseteq \textit{noacks}) \land (\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks} \setminus (\textit{failureacks?} \cup \textit{repairs?}))) \land \\
\qquad (((\textit{failureacks?} \subseteq \textit{noacks}) \land \neg(\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks} \setminus \textit{failureacks?})) \land \\
\qquad ((\neg(\textit{failureacks?} \subseteq \textit{noacks}) \land (\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks} \setminus \textit{repairs?})) \land \\
\qquad ((\neg(\textit{failureacks?} \subseteq \textit{noacks}) \land \neg(\textit{repairs?} \subseteq \textit{failures})) \\
\qquad\quad \Rightarrow (\textit{noacks}' = \textit{newnoacks})) \\
\quad) \\
) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{Equipment} == (\ \textit{QLowerBoundValveOpen} \lor \textit{QLowerBoundValveClosed}\ ) \land \\
\quad \textit{QUpperBound} \land \textit{VLowerBound} \land \textit{VUpperBound} \land \\
\quad \textit{ExpectedPumpStates} \land \textit{EquipmentFailures}
\end{array}
$$

In *InitEquipment*, expected pump and pump controller states and valve state are initialised as well.

$$
\begin{array}{l}
\textit{InitEquipment} == \textit{Equipment0}\,' \land \textit{InitQSensor} \land \textit{InitVSensor} \land \\
\quad \textit{InitPumpCtrSystem} \land \textit{InitCValues} \land \textit{InitFailures} \land \\
\quad \textit{InitExpectedPumpStates} \land \textit{InitValve}
\end{array}
$$

I.1.2.7 Repair Failed Equipments

This is a newly added section to repair equipments according to input *repairs*?. For *QSensor*, if it is repaired, then its *qst* will be *sokay*. Otherwise it stays unchanged.

$$
\begin{array}{l}
\textit{RepairQSensor} \\
\hline
\Delta \textit{QSensor} \\
\textit{repairs?} : \mathbf{P}\ \textit{UnitFailure} \\
\hline
\textit{qa\_1}' = \textit{qa\_1} \\
\textit{qa\_2}' = \textit{qa\_2} \\
\textit{qfail} \in \textit{repairs?} \Rightarrow \textit{qst}' = \textit{sokay} \\
\textit{qfail} \notin \textit{repairs?} \Rightarrow \textit{qst}' = \textit{qst} \\
\hline
\end{array}
$$

For *VSensor*, if it is repaired, then its *vst* will be *sokay*. Otherwise it stays unchanged.

$$\begin{array}{|l}
\textit{RepairVSensor} \\
\hline
\Delta \textit{VSensor} \\
\textit{repairs?} : \mathbb{P}\ \textit{UnitFailure} \\
\hline
\textit{va\_1}' = \textit{va\_1} \\
\textit{va\_2}' = \textit{va\_2} \\
\textit{vfail} \in \textit{repairs?} \Rightarrow \textit{vst}' = \textit{sokay} \\
\textit{vfail} \notin \textit{repairs?} \Rightarrow \textit{vst}' = \textit{vst} \\
\hline
\end{array}$$

If a pump controller is repaired, its state will be *pcflow* if current pump state is *popen*, or its state will be *pcnoflow* if current pump state is not *popen*.

$$\begin{array}{|l}
\textit{RepairAPumpCtr} \\
\hline
\Delta \textit{PumpCtr} \\
\hline
\textit{pst}' = \textit{pst} \\
\textit{pst} = \textit{popen} \Rightarrow \textit{pcst}' = \textit{pcflow} \\
\textit{pst} \neq \textit{popen} \Rightarrow \textit{pcst}' = \textit{pcnoflow} \\
\hline
\end{array}$$

If a pump is repaired, its state will be *pclosed* and its pump controller state stays unchanged.

$$\begin{array}{|l}
\textit{RepairAPump} \\
\hline
\Delta \textit{PumpCtr} \\
\hline
\textit{pst}' = \textit{pclosed} \\
\textit{pcst}' = \textit{pcst} \\
\hline
\end{array}$$

If both a pump and its controller are repaired, then the pump will be *pclosed* and its controller will be *pcnoflow*.

$$\begin{array}{|l}
\textit{RepairPumpCtrAndPump} \\
\hline
\Delta \textit{PumpCtr} \\
\hline
\textit{pst}' = \textit{pclosed} \\
\textit{pcst}' = \textit{pcnoflow} \\
\hline
\end{array}$$

The schema *RepairPumps* repairs all pumps and their controllers according to input *repairs?*.

$$\begin{array}{|l}
\textit{RepairPumps} \\
\hline
\Delta \textit{PumpCtrSystem} \\
\textit{repairs?} : \mathbb{P}\ \textit{UnitFailure} \\
\hline
\forall\, i : \textit{PumpIndex} \bullet \\
\quad \exists\, \textit{PumpCtr} ; \textit{PumpCtr}\,' \bullet (\\
\qquad (\theta\, \textit{PumpCtr}\,' = \textit{pumpctr}'\ i) \land (\theta\, \textit{PumpCtr} = \textit{pumpctr}\ i) \land \\
\qquad ((\textit{pfail}\ i \in \textit{repairs?} \land \textit{pcfail}\ i \notin \textit{repairs?}) \\
\qquad\quad \Rightarrow \textit{RepairAPump}) \land \\
\qquad ((\textit{pfail}\ i \notin \textit{repairs?} \land \textit{pcfail}\ i \in \textit{repairs?}) \\
\qquad\quad \Rightarrow \textit{RepairAPumpCtr}) \land \\
\qquad ((\textit{pfail}\ i \in \textit{repairs?} \land \textit{pcfail}\ i \in \textit{repairs?}) \\
\qquad\quad \Rightarrow \textit{RepairPumpCtrAndPump}) \land \\
\qquad ((\textit{pfail}\ i \notin \textit{repairs?} \land \textit{pcfail}\ i \notin \textit{repairs?}) \\
\qquad\quad \Rightarrow \theta\, \textit{PumpCtr}\,' = \theta\, \textit{PumpCtr}) \\
\quad) \\
\hline
\end{array}$$

The *RepairEquipments* tries to repair all equipments according to input *repairs*?. If *repairs*? are accepted, all equipments will be repaired. Otherwise, all equipments will stay unchanged.

$$
\begin{array}{l}
RepairEquipments == \\
\quad (RepairsExpected[repairs?/repairs] \land \\
\quad\quad RepairPumps \land RepairQSensor \land RepairVSensor \\
\quad) \lor \\
\quad ((\lnot RepairsExpected[repairs?/repairs]) \land \\
\quad\quad \Xi PumpCtrSystem \land \Xi QSensor \land \Xi VSensor \\
\quad)
\end{array}
$$

A *emergencyCond* state is introduced to indicate if both input *repairs*? and *failureacks*? are accepted or not. It is set to 1 if there is unaccepted *repairs*? or *failureacks*?, or both. Otherwise, it is set to 0. This update happens in the beginning of each cycle and the value is used in the later of the cycle.

$$
\begin{array}{l}
EmergenyCond == [\, emergencyCond : \{0, 1\} \,] \\
MarkEmergencyCond == [\, \Delta EmergenyCond \mid emergencyCond' = 1 \,] \\
ClearEmergencyCond == [\, \Delta EmergenyCond \mid emergencyCond' = 0 \,] \\
EvalRepairFailureAck == \\
\quad (RepairsExpected[repairs?/repairs] \land \\
\quad\quad FailuresExpected[failureacks?/failureacks] \land \\
\quad\quad ClearEmergencyCond \\
\quad) \lor \\
\quad ((\lnot RepairsExpected[repairs?/repairs] \lor \\
\quad\quad \lnot FailuresExpected[failureacks?/failureacks]) \\
\quad\quad \land MarkEmergencyCond \\
\quad)
\end{array}
$$

I.1.2.8 Input messages

$$
\begin{array}{l}
InputSignal ::= \\
\quad stop \mid steamBoilerWaiting \mid physicalUnitsReady \mid transmissionFailure
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{UnitState} \\
\hline
pumpState : PumpIndex \rightarrow InputPState \\
pumpCtrState : PumpIndex \rightarrow InputPCState \\
q, v : NUMS \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{InputMsg} \\
\hline
signals : \mathbf{P}\ InputSignal \\
UnitState \\
failureacks, repairs : \mathbf{P}\ UnitFailure \\
\hline
\end{array}
$$

I.1.2.9 Analysing messages

The input value $x?$ should be checked against calculated values c_1 and c_2, instead of adjusted values a_1 and a_2.

$$
\begin{array}{l}
Expected == [\, x?, c\_1, c\_2 : NUMS \mid c\_1 \leq x? \leq c\_2 \,] \\
Unexpected == \lnot Expected
\end{array}
$$

$$Sensor == [\,\Delta\, Unit[SState]\,;\, c\_1\,,\, c\_2\,,\, c\_1'\,,\, c\_2'\,,\, x? : NUMS\,]$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CheckAndAdjustSensor \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
Sensor \\
\hline
Expected \Rightarrow st' = st \\
Unexpected \Rightarrow st' = sfailed \\
st' = sokay \Rightarrow a\_1' = x? \land a\_2' = x? \\
st' = sfailed \Rightarrow a\_1' = c\_1 \land a\_2' = c\_2 \\
\hline
\end{array}$$

$$\begin{array}{l}
CheckAndAdjustQ == QSensor \land \\
\quad CheckAndAdjustSensor[\\
\quad\quad q?/x?, qa\_1\,/\,a\_1\,,\, qa\_2\,/\,a\_2\,,\, qc\_1\,/\,c\_1\,,\, qc\_2\,/\,c\_2\,,\, qst/st, \\
\quad\quad qa\_1'\,/\,a\_1'\,,\, qa\_2'\,/\,a\_2'\,,\, qc\_1'\,/\,c\_1'\,,\, qc\_2'\,/\,c\_2'\,,\, qst'/st']
\end{array}$$

$$\begin{array}{l}
CheckAndAdjustV == VSensor \land \\
\quad CheckAndAdjustSensor[\\
\quad\quad v?/x?, va\_1\,/\,a\_1\,,\, va\_2\,/\,a\_2\,,\, vc\_1\,/\,c\_1\,,\, vc\_2\,/\,c\_2\,,\, vst/st, \\
\quad\quad va\_1'\,/\,a\_1'\,,\, va\_2'\,/\,a\_2'\,,\, vc\_1'\,/\,c\_1'\,,\, vc\_2'\,/\,c\_2'\,,\, vst'/st']
\end{array}$$

The *ExpectedPumpStateTBD* checks if the expected pumps and their controllers state are undetermined. This happens in the initialisation stage when the expected states are unknown. And we indicate this in *InitExpectedPumpStates*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ExpectedPumpStateTBD \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
exppst : InputPState \\
exppcst : InputPCState \\
\hline
exppst = pclosed \\
exppcst = pcflow \\
\hline
\end{array}$$

If expected pump states are unknown, we adjust pumps and their controllers states according to input states only and will not check expected pump states.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CheckAndAdjustPumpTBD \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta PumpCtr \\
pst?, exppst : InputPState \\
pcst?, exppcst : InputPCState \\
\hline
((pst? = popen \land pcst? = pcflow) \lor (pst? = pclosed \land pcst? = pcnoflow)) \\
\quad\quad \Rightarrow (pst' = pst? \land pcst' = pcst?) \\
(pst? = popen \land pcst? = pcnoflow) \Rightarrow (pst' = pfailed \land pcst' = pcnoflow) \\
(pst? = pclosed \land pcst? = pcflow) \Rightarrow (pst' = pfailed \land pcst' = pcflow) \\
\hline
\end{array}$$

However, if expected pump states are valid, we adjust pumps and their controllers states according to input and expected pump states together.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CheckAndAdjustPump}} \\
\hline
\Delta \mathit{PumpCtr} \\
pst?, exppst : \mathit{InputPState} \\
pcst?, exppcst : \mathit{InputPCState} \\
\hline
((pst = pfailed \land pst' = pst) \lor \\
\qquad (pst \neq pfailed \land \\
\qquad\qquad (pst? = exppst \Rightarrow pst' = pst?) \land \\
\qquad\qquad (pst? \neq exppst \Rightarrow pst' = pfailed) \\
\qquad) \\
) \\
((pcst = pcfailed \land pcst' = pcst) \lor \\
\qquad (pcst \neq pcfailed \land \\
\qquad\qquad (pcst? = exppcst \Rightarrow pcst' = pcst?) \land \\
\qquad\qquad (pcst? \neq exppcst \Rightarrow pcst' = pcfailed) \\
\qquad) \\
) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PromotePumpCheck}} \\
\hline
\Delta \mathit{PumpCtr} \\
\Delta \mathit{PumpCtrSystem} \\
\mathit{ExpectedPumpStates} \\
pst?, exppst : \mathit{InputPState} \\
pcst?, exppcst : \mathit{InputPCState} \\
pumpState? : \mathit{PumpIndex} \rightarrow \mathit{InputPState} \\
pumpCtrState? : \mathit{PumpIndex} \rightarrow \mathit{InputPCState} \\
i : \mathit{PumpIndex} \\
\hline
\theta\, \mathit{PumpCtr} = pumpctr\ i \\
\theta\, \mathit{PumpCtr}\,' = pumpctr'\ i \\
pst? = pumpState?\ i \\
pcst? = pumpCtrState?\ i \\
exppst = expectedp\ i \\
exppcst = expectedpc\ i \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{SetPumpCtr} == \forall i : \mathit{PumpIndex} \bullet \\
\qquad \exists\, \mathit{PumpCtr} ; \mathit{PumpCtr}\,' ; pst?, exppst : \mathit{PState} ; pcst?, exppcst : \mathit{PCState} \bullet \\
\qquad\qquad (\mathit{PromotePumpCheck} \land \\
\qquad\qquad\qquad ((\mathit{CheckAndAdjustPumpTBD} \land \mathit{ExpectedPumpStateTBD}) \lor \\
\qquad\qquad\qquad\qquad (\mathit{CheckAndAdjustPump} \land \neg \mathit{ExpectedPumpStateTBD}) \\
\qquad\qquad\qquad) \\
\qquad\qquad)
\end{array}
$$

The original predicate of *StopPresent* is correct. Just because we introduce *NUMS* for animation, the predicate of *StopPresent* is modified too.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StopPresent}} \\
\hline
signals? : \mathbf{P}\ \mathit{InputSignal} \\
stops, stops' : \mathit{NUMS} \\
\hline
stop \in signals? \\
((stops + 1 > \mathit{MAX\_NUM} \land stops' = stops) \lor (stops' = stops + 1)) \\
\hline
\end{array}
$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StopNotPresent}} \\
\hline
\mathit{signals}? : \mathbf{P}\ \mathit{InputSignal} \\
\mathit{stops}, \mathit{stops}' : \mathit{NUMS} \\
\hline
\mathit{stop} \notin \mathit{signals}? \land \mathit{stops} < 3 \\
\mathit{stops}' = 0 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{TooManyStops}} \\
\hline
\mathit{signals}? : \mathbf{P}\ \mathit{InputSignal} \\
\mathit{stops}, \mathit{stops}' : \mathit{NUMS} \\
\hline
\mathit{stop} \notin \mathit{signals}? \land \mathit{stops} \geq 3 \\
\mathit{stops}' = \mathit{stops} \\
\hline
\end{array}$$

$$\mathit{AdjustStops} == \mathit{StopPresent} \lor \mathit{StopNotPresent} \lor \mathit{TooManyStops}$$

I.1.2.10 The Process

$$\begin{array}{l}
\textbf{channel}\ \mathit{levelbelowmin}, \mathit{levelabovemax} \\
\textbf{channel}\ \mathit{emergencystop}, \mathit{cfailures}, \mathit{levelokay}, \mathit{nonqfailure} : \mathbb{B} \\
\textbf{channel}\ \mathit{physicalunitsready}, \mathit{qfailure}, \mathit{sbwaiting}, \mathit{vzero} : \mathbb{B}
\end{array}$$

For animation purpose, *input* has been split into seven small channels: $input1$, $input2$, $input3$, $input4$, $input5$, $input6$, and $input7$.

$$\begin{array}{l}
\textbf{channel}\ \mathit{input}1 : (\mathbf{P}\ \mathit{InputSignal}) \\
\textbf{channel}\ \mathit{input}2 : (\mathit{PumpIndex} \to \mathit{InputPState}) \\
\textbf{channel}\ \mathit{input}3 : (\mathit{PumpIndex} \to \mathit{InputPCState}) \\
\textbf{channel}\ \mathit{input}4 : (\mathit{NUMS}) \\
\textbf{channel}\ \mathit{input}5 : (\mathit{NUMS}) \\
\textbf{channel}\ \mathit{input}6 : (\mathbf{P}\ \mathit{UnitFailure}) \\
\textbf{channel}\ \mathit{input}7 : (\mathbf{P}\ \mathit{UnitFailure}) \\
\textbf{channel}\ \mathit{startexec}
\end{array}$$

$$\textbf{channel}\ \mathit{failuresrepairs} : (\mathbf{P}\ \mathit{UnitFailure}) \times (\mathbf{P}\ \mathit{UnitFailure})$$

$$\begin{array}{l}
\textbf{channel}\ \mathit{pumps} : (\mathit{PumpIndex} \to \mathit{InputPState}) \times \mathit{VAction} \\
\textbf{channelset}\ \mathit{Information} == \\
\quad \{\!|\ \mathit{emergencystop}, \mathit{cfailures}, \mathit{levelabovemax}, \mathit{levelbelowmin}, \mathit{levelokay}, \\
\qquad \mathit{nonqfailure}, \mathit{physicalunitsready}, \mathit{qfailure}, \mathit{sbwaiting}, \mathit{vzero}\ |\!\}
\end{array}$$

$$\textbf{process}\ \mathit{Analyser} \mathrel{\widehat=} \textbf{begin}$$

$$\begin{array}{l}
\textbf{state}\ \mathit{AnalyserState} == [\, \mathit{Equipment}0 ; \mathit{Failures} ; \mathit{InputMsg}; \\
\qquad \mathit{stops} : \mathit{NUMS} ; \mathit{signalhistory} : \mathbf{P}\ \mathit{InputSignal} ; \mathit{EmergenyCond}\,]
\end{array}$$

$$\begin{array}{l}
\mathit{StopSignalHis} == [\, \mathit{stops} : \mathit{NUMS} ; \mathit{signalhistory} : \mathbf{P}\ \mathit{InputSignal}\,] \\
\mathit{PumpOp} == \Xi \mathit{QSensor} \land \Xi \mathit{VSensor} \land \Xi \mathit{Valve} \land \Xi \mathit{CValues} \land \\
\qquad \Xi \mathit{Failures} \land \Xi \mathit{ExpectedPumpStates} \land \Xi \mathit{InputMsg} \land \\
\qquad \Xi \mathit{StopSignalHis} \land \Xi \mathit{EmergenyCond}
\end{array}$$

For *InputMsg*, its initial value can be arbitrarily chosen and it will not have impacts on the behaviour of the program. To ease model checking, we set a specific initial value in *InitAnalyserState*.

$$
\begin{aligned}
&InitAnalyserState == [\, AnalyserState' \mid \\
&\quad InitEquipment \land stops' = 0 \land signalhistory' = \varnothing \land \\
&\quad \theta\, InputMsg' = (\mathbf{let}\ signals == \varnothing[InputSignal]; \\
&\qquad pumpState == \\
&\qquad\quad \{1 \mapsto pclosed, 2 \mapsto pclosed, 3 \mapsto pclosed, 4 \mapsto pclosed\}; \\
&\qquad pumpCtrState == \\
&\qquad\quad \{1 \mapsto pcnoflow, 2 \mapsto pcnoflow, 3 \mapsto pcnoflow, 4 \mapsto pcnoflow\}; \\
&\qquad q == 0\,;\, v == 0\,;\, failureacks == \varnothing[UnitFailure]; \\
&\qquad repairs == \varnothing[UnitFailure] \bullet \\
&\qquad \theta\, InputMsg) \\
&\quad \land\ emergencyCond' = 0\,]
\end{aligned}
$$

$$
\begin{aligned}
&Analyse == \\
&\quad [\,\Delta AnalyserState\,;\, InputMsg\,? \mid \theta\, InputMsg' = \theta\, InputMsg\,? \land \\
&\quad CheckAndAdjustQ \land CheckAndAdjustV \land AdjustStops \land \\
&\quad signalhistory' = signalhistory \cup signals? \land \\
&\quad UpdateFailuresAck \land \Xi PumpCtrSystem \land \Xi ExpectedPumpStates \land \\
&\quad \Xi Valve \land Equipment' \land \Xi EmergenyCond\,]
\end{aligned}
$$

In its predicate, $N\_1 < qa\_2$ should be $N\_2 < qa\_2$.

$$
\begin{aligned}
&DangerZone == [\, AnalyserState \mid qa\_1 \geq M\_1 \land qa\_2 \leq M\_2 \\
&\quad \Rightarrow qa\_1 < N\_1 \land N\_2 < qa\_2\,]
\end{aligned}
$$

Instead of checking $\neg RepairsExpected \lor \neg FailuresExpected$, we check *emergencyCond*, because in the later stage, the *failures* and *noacks* have been updated and not original values. Therefore, it is wrong to check *repairs*? and *failureacks*? against updated *failures* and *noacks*.

$$
\begin{aligned}
&EmergencyStopCond == [\, AnalyserState \mid \\
&\quad stops \geq 3 \lor DangerZone \lor emergencyCond = 1 \lor \\
&\quad transmissionFailure \in signals\,]
\end{aligned}
$$

$$
\begin{aligned}
&LevelBelowMin == [\, AnalyserState \mid M\_1 \leq qa\_1 < N\_1 \land qa\_2 \leq N\_2\,] \\
&LevelAboveMax == [\, AnalyserState \mid N\_1 \leq qa\_1 \land N\_2 < qa\_2 \leq M\_2\,] \\
&LevelInRange == [\, AnalyserState \mid N\_1 \leq qa\_1 \land qa\_2 \leq N\_2\,]
\end{aligned}
$$

$$
\begin{aligned}
&RateZero == [\, VSensor \mid va\_1 = 0 \land va\_2 = 0\,] \\
&AllPhysicalUnitsOkay == \\
&\quad [\, AnalyserState \mid \neg QFailed \land \neg VFailed \land \neg PFailed \land \neg PCFailed\,] \\
&OtherPhysicalUnitsFail == \neg QFailed \land \neg AllPhysicalUnitsOkay
\end{aligned}
$$

$$
\begin{aligned}
&SteamBoilerWaiting == \\
&\quad [\, AnalyserState \mid steamBoilerWaiting \in signalhistory\,] \\
&PhysicalUnitsReady == \\
&\quad [\, AnalyserState \mid physicalUnitsReady \in signalhistory\,]
\end{aligned}
$$

HandleRepair, as a schema expression, is added to repair equipments.

$$
\begin{array}{l}
\quad \mathit{HandleRepair} == \mathit{RepairEquipments} \land \mathit{EvalRepairFailureAck} \land \Xi \mathit{CValues} \\
\qquad \land\ \Xi \mathit{Failures} \land \Xi \mathit{InputMsg} \land \Xi \mathit{StopSignalHis} \land \\
\qquad \Xi \mathit{Valve} \land \Xi \mathit{ExpectedPumpStates}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{AnalyserCycle} \mathrel{\widehat=} \mathit{startcycle} \to \mathit{input1?signals} \to \mathit{input2?pumpState} \to \\
\qquad \mathit{input3?pumpCtrState} \to \mathit{input4?q} \to \mathit{input5?v} \to \\
\qquad \mathit{input6?failureacks} \to \mathit{input7?repairs} \to \\
\qquad (\left(\mathit{HandleRepair}\right) ; \left(\mathit{SetPumpCtr} \land \mathit{PumpOp}\right); \\
\qquad\quad \left(\mathit{Analyse}\right) ;\ \mathit{startexec} \to \mathit{InfoService})
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{PumpOp2} == \Xi \mathit{QSensor} \land \Xi \mathit{VSensor} \land \Xi \mathit{CValues} \land \\
\qquad \Xi \mathit{Failures} \land \Xi \mathit{InputMsg} \land \Xi \mathit{StopSignalHis} \land \\
\qquad \Xi \mathit{EmergenyCond} \\
\quad \mathit{SetExpectedPumpState} == \\
\qquad \mathit{CalcExpectedPumpState} \land \mathit{SetValveState} \land \mathit{PumpOp2}
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{InfoService} \mathrel{\widehat=} (\mathit{OfferInformation} ;\ \mathit{InfoService}) \mathrel{\Box} \\
\qquad \mathit{failuresrepairs\,!noacks!repairs} \to \mathit{pumps\,?pumpstate?vstate} \to \\
\qquad \left(\mathit{SetExpectedPumpState}\right) ; \mathit{AnalyserCycle} \\
\quad \mathit{OfferInformation} \mathrel{\widehat=} \\
\qquad \mathit{emergencystop.EmergencyStopCond} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{sbwaiting.SteamBoilerWaiting} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{vzero.RateZero} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad (\mathit{LevelBelowMin}) \mathrel{\&}\ \mathit{levelbelowmin} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad (\mathit{LevelAboveMax}) \mathrel{\&}\ \mathit{levelabovemax} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{levelokay.LevelInRange} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{physicalunitsready.PhysicalUnitsReady} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{cfailures.}(\lnot \mathit{AllPhysicalUnitsOkay}) \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{qfailure.QFailed} \to \mathbf{Skip} \\
\qquad \Box \\
\qquad \mathit{nonqfailure.OtherPhysicalUnitsFail} \to \mathbf{Skip}
\end{array}
$$

$$
\quad \bullet \left(\mathit{InitAnalyserState}\right) ;\ \mathit{AnalyserCycle}
$$

end

$$
\begin{array}{l}
\mathbf{channelset}\ \ \mathit{TAnalyserInterface} == \{\!|\ \mathit{startcycle}\ |\!\} \\
\mathbf{process}\ \ \mathit{TAnalyser} \mathrel{\widehat=} \\
\quad \mathit{Timer} \mathrel{[\![} \mathit{TAnalyserInterface} \mathrel{]\!]} \mathit{Analyser} \setminus \mathit{TAnalyserInterface}
\end{array}
$$

I.1.3 Controller

$$Mode ::= initialisation \mid normal \mid degraded \mid rescue \mid emergencyStop$$
$$Nonemergency == \{initialisation, normal, degraded, rescue\}$$

$$\begin{array}{l}
\mathbf{channel}\ startpumps, stoppumps, openvalve, closevalve, sendprogready \\
\mathbf{channel}\ reportmode : Mode \\
\mathbf{channel}\ startreport, endreport \\
\mathbf{channelset}\ Reports == \\
\quad \{\!|\ startpumps, stoppumps, openvalve, closevalve, sendprogready\ |\!\} \\
\mathbf{channelset}\ TAControllerInterface == \{\!|\ startexec\ |\!\} \cup Information
\end{array}$$

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

$$\begin{array}{l}
\mathbf{state}\ ModeState == [\, mode : Mode\,] \\
InitController == [\, ModeState' \mid mode' = initialisation\,] \\
EnterMode \mathrel{\widehat=} m : Mode \bullet reportmode\,!m \rightarrow mode := m
\end{array}$$

In *emergencyStop* mode, it is not necessary to adjust level *AdjustLevel* and just end report by *endreport*.

$$\begin{array}{l}
ControllerCycle \mathrel{\widehat=} startexec \rightarrow startreport \rightarrow NewModeAnalysis; \\
\quad ((mode \neq emergencyStop)\ \&\ AdjustLevel\ \Box \\
\quad (mode = emergencyStop)\ \&\ Skip); \\
\quad endreport \rightarrow ControllerCycle \\
NewModeAnalysis \mathrel{\widehat=} emergencystop.\mathbf{True} \rightarrow EnterMode\,(emergencyStop) \\
\quad \Box\ emergencystop.\mathbf{False} \rightarrow (\\
\qquad (mode = initialisation)\ \&\ \ InitModeAnalysis \\
\qquad \Box\ (mode = normal)\ \&\ \ NormalModeAnalysis \\
\qquad \Box\ (mode = degraded)\ \&\ \ DegradedModeAnalysis \\
\qquad \Box\ (mode = rescue)\ \&\ \ RescueModeAnalysis \\
\qquad \Box\ ((mode \notin Mode \setminus \{emergencyStop\}))\ \&\ \ \mathbf{Skip} \\
\quad)
\end{array}$$

$$\begin{array}{l}
InitModeAnalysis \mathrel{\widehat=} \\
\quad sbwaiting.\mathbf{True} \rightarrow \\
\qquad (\, vzero.\mathbf{True} \rightarrow \\
\qquad\quad (\, qfailure.\mathbf{False} \rightarrow \\
\qquad\qquad (\, physicalunitsready.\mathbf{True} \rightarrow \\
\qquad\qquad\quad (\, levelokay.\mathbf{True} \rightarrow \\
\qquad\qquad\qquad (\, cfailures.\mathbf{False} \rightarrow EnterMode\,(normal)\ \Box \\
\qquad\qquad\qquad cfailures.\mathbf{True} \rightarrow EnterMode\,(degraded)\,)\ \Box \\
\qquad\qquad\quad levelokay.\mathbf{False} \rightarrow EnterMode\,(emergencyStop)\,)\ \Box \\
\qquad\qquad physicalunitsready.\mathbf{False} \rightarrow \\
\qquad\qquad\quad (\, levelokay.\mathbf{True} \rightarrow \\
\qquad\qquad\qquad sendprogready \rightarrow \mathbf{Skip}\ \Box \\
\qquad\qquad\quad levelokay.\mathbf{False} \rightarrow \mathbf{Skip}\,)\,)\ \Box \\
\qquad\quad qfailure.\mathbf{True} \rightarrow EnterMode\,(emergencyStop)\,)\ \Box \\
\qquad vzero.\mathbf{False} \rightarrow EnterMode\,(emergencyStop)\,)\ \Box \\
\quad sbwaiting.\mathbf{False} \rightarrow \mathbf{Skip}
\end{array}$$

```
    $NormalModeAnalysis \mathrel{\widehat=}$
        $cfailures.\mathbf{False} \rightarrow \mathbf{Skip} \Box$
        $qfailure.\mathbf{True} \rightarrow EnterMode\,(rescue) \Box$
        $nonqfailure.\mathbf{True} \rightarrow EnterMode\,(degraded)$

    $DegradedModeAnalysis \mathrel{\widehat=}$
        $qfailure.\mathbf{False} \rightarrow$
            $(cfailures.\mathbf{True} \rightarrow \mathbf{Skip} \Box$
            $cfailures.\mathbf{False} \rightarrow EnterMode\,(normal)\,)$
        $\Box\ qfailure.\mathbf{True} \rightarrow EnterMode\,(rescue)$

    $RescueModeAnalysis \mathrel{\widehat=}$
        $qfailure.\mathbf{True} \rightarrow \mathbf{Skip} \Box$
        $qfailure.\mathbf{False} \rightarrow ($
            $cfailures.\mathbf{False} \rightarrow EnterMode\,(normal)$
            $\Box\ cfailures.\mathbf{True} \rightarrow EnterMode\,(degraded)\,)$

    $AdjustLevel \mathrel{\widehat=} levelbelowmin \rightarrow RaiseLevel \Box$
        $levelabovemax \rightarrow ReduceLevel \Box$
        $levelokay.\mathbf{True} \rightarrow RetainLevel$

    $RaiseLevel \mathrel{\widehat=} StartPumps;$
        $\mathbf{if}\ mode = initialisation \longrightarrow CloseValve$
        $[\!]\,mode \neq initialisation \longrightarrow \mathbf{Skip}$
        $\mathbf{fi}$

    $ReduceLevel \mathrel{\widehat=} StopPumps;$
        $\mathbf{if}\ mode = initialisation \longrightarrow OpenValve$
          $[\!]\,mode \neq initialisation \longrightarrow \mathbf{Skip}$
        $\mathbf{fi}$

    $RetainLevel \mathrel{\widehat=} StopPumps;$
        $\mathbf{if}\ mode = initialisation \longrightarrow CloseValve$
          $[\!]\,mode \neq initialisation \longrightarrow \mathbf{Skip}$
        $\mathbf{fi}$

    $StartPumps \mathrel{\widehat=} startpumps \rightarrow \mathbf{Skip}$
    $StopPumps \mathrel{\widehat=} stoppumps \rightarrow \mathbf{Skip}$
    $OpenValve \mathrel{\widehat=} openvalve \rightarrow \mathbf{Skip}$
    $CloseValve \mathrel{\widehat=} closevalve \rightarrow \mathbf{Skip}$


    $\bullet\ \big(InitController\big)\ ;\ ControllerCycle$


$\mathbf{end}$

$\mathbf{process}\ \ TAController \mathrel{\widehat=}$
    $(TAnalyser \llbracket\ TAControllerInterface\ \rrbracket\ Controller) \setminus TAControllerInterface$
```

I.1.4 Reporter

$$
\begin{aligned}
&OutputSignal ::= programReady \mid openValve \mid closeValve \mid \\
&\quad levelFailureDetection \mid steamFailureDetection \mid \\
&\quad levelRepairedAcknowledgement \mid steamRepairedAcknowledgement
\end{aligned}
$$

$$
\begin{array}{|l}
\hline OutputMsg \\
mode : Mode \\
signals : \mathbf{P}\ OutputSignal \\
pumpState : PumpIndex \rightarrow InputPState \\
pumpFailureDetection : \mathbf{P}\ UnitFailure \\
pumpCtrFailureDetection : \mathbf{P}\ UnitFailure \\
pumpRepairedAcknowledgement : \mathbf{P}\ UnitFailure \\
pumpCtrRepairedAcknowledgement : \mathbf{P}\ UnitFailure \\
\hline
\end{array}
$$

Similar to the *input* channel, the *output* channel is split too.

$$
\begin{aligned}
&\mathbf{channel}\ \ output1 : Mode \\
&\mathbf{channel}\ \ output2 : (\mathbf{P}\ OutputSignal) \\
&\mathbf{channel}\ \ output3 : (PumpIndex \rightarrow InputPState) \\
&\mathbf{channel}\ \ output4 : (\mathbf{P}\ UnitFailure) \\
&\mathbf{channel}\ \ output5 : (\mathbf{P}\ UnitFailure) \\
&\mathbf{channel}\ \ output6 : (\mathbf{P}\ UnitFailure) \\
&\mathbf{channel}\ \ output7 : (\mathbf{P}\ UnitFailure)
\end{aligned}
$$

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

$$\mathbf{state}\ \ ReporterState == [\, OutputMsg ; valveSt : VAction \mid true \,]$$

Similar to the *Timer* process and initial value of *InputMsg*, we initialise *OutputMsg* as well though its initial value can be arbitrarily chosen.

$$
\begin{aligned}
&InitReporter == [\, ReporterState' \mid valveSt' = VNoChange \land \\
&\quad \theta\, OutputMsg' = \\
&\quad (\mathbf{let}\ mode == initialisation ; signals == \varnothing[OutputSignal]; \\
&\qquad pumpState == \{1 \mapsto pclosed, 2 \mapsto pclosed, \\
&\qquad\quad 3 \mapsto pclosed, 4 \mapsto pclosed\}; \\
&\qquad pumpFailureDetection == \varnothing[UnitFailure]; \\
&\qquad pumpCtrFailureDetection == \varnothing[UnitFailure]; \\
&\qquad pumpRepairedAcknowledgement == \varnothing[UnitFailure]; \\
&\qquad pumpCtrRepairedAcknowledgement == \varnothing[UnitFailure] \\
&\qquad \bullet\ \theta\, OutputMsg) \,]
\end{aligned}
$$

$$
\begin{aligned}
&ReportService \mathrel{\widehat=} GatherReports ; ReportService \ \Box \\
&\quad reportmode.emergencyStop \rightarrow mode := emergencyStop ; TidyUp \ \Box \\
&\quad TidyUp
\end{aligned}
$$

This schema is used to update *OutputMsg* according to the inputs *noacks* and *repairs* from the *Analyser* process.

$$
\begin{array}{l}
FailuresRepairs == [\,\Delta ReporterState\ ;\ noacks? : (\mathbb{P}\ UnitFailure);\\
\quad repairs? : (\mathbb{P}\ UnitFailure)\ |\\
\quad (signals' = signals \cup\\
\quad\quad (\mathbf{if}(qfail \in noacks?)\ \mathbf{then}\ \{levelFailureDetection\}\ \mathbf{else}\ \emptyset) \cup\\
\quad\quad (\mathbf{if}\ vfail \in noacks?\ \mathbf{then}\ \{steamFailureDetection\}\ \mathbf{else}\ \emptyset) \cup\\
\quad\quad (\mathbf{if}\ qfail \in repairs?\ \mathbf{then}\ \{levelRepairedAcknowledgement\}\\
\quad\quad\quad \mathbf{else} \emptyset) \cup\\
\quad\quad (\mathbf{if}\ vfail \in repairs?\ \mathbf{then}\ \{steamRepairedAcknowledgement\}\\
\quad\quad\quad \mathbf{else} \emptyset)) \land\\
\quad pumpFailureDetection' =\\
\quad\quad noacks? \cap \{i : PumpIndex \bullet pfail\ i\} \land\\
\quad pumpCtrFailureDetection' =\\
\quad\quad noacks? \cap \{i : PumpIndex \bullet pcfail\ i\} \land\\
\quad pumpRepairedAcknowledgement' =\\
\quad\quad repairs? \cap \{i : PumpIndex \bullet pfail\ i\} \land\\
\quad pumpCtrRepairedAcknowledgement' =\\
\quad\quad repairs? \cap \{i : PumpIndex \bullet pcfail\ i\} \land\\
\quad mode' = mode \land valveSt' = valveSt \land pumpState' = pumpState\,]
\end{array}
$$

$$
\begin{array}{l}
TidyUp \mathrel{\widehat=} endreport \rightarrow failuresrepairs\,?noacks?repairs \rightarrow \big(FailuresRepairs\big);\\
\quad output1!mode \rightarrow output2!signals \rightarrow output3!pumpState \rightarrow\\
\quad output4!pumpFailureDetection \rightarrow output5!pumpCtrFailureDetection \rightarrow\\
\quad output6!pumpRepairedAcknowledgement \rightarrow\\
\quad output7!pumpCtrRepairedAcknowledgement \rightarrow\\
\quad pumps\,!pumpState!valveSt \rightarrow \mathbf{Skip}\\
GatherReports \mathrel{\widehat=} \Box\ m : Nonemergency \bullet reportmode.m \rightarrow mode := m\\
\quad\quad \Box\\
\quad\quad sendprogready \rightarrow signals := signals \cup \{programReady\}\\
\quad\quad \Box\\
\quad\quad startpumps \rightarrow pumpState := PumpIndex \times \{popen\}\\
\quad\quad \Box\\
\quad\quad stoppumps \rightarrow pumpState := PumpIndex \times \{pclosed\}\\
\quad\quad \Box\\
\quad\quad openvalve \rightarrow signals, valveSt := signals \cup \{openValve\}, openv\\
\quad\quad \Box\\
\quad\quad closevalve \rightarrow signals, valveSt := signals \cup \{closeValve\}, closev
\end{array}
$$

$$
\bullet\ \mu X \bullet startreport \rightarrow \big(InitReporter\big)\ ;\ ReportService\ ;\ X
$$

end

$$
\begin{array}{l}
\mathbf{channelset}\ \ TACReporterInterface ==\\
\quad \{\!|\ startpumps, stoppumps, openvalve, closevalve, sendprogready,\\
\quad\quad startreport, reportmode, endreport, failuresrepairs, pumps\ |\!\}\\
\mathbf{process}\ \ TACReporter \mathrel{\widehat=}\\
\quad (TAController\\
\quad\quad [\![TACReporterInterface]\!]\\
\quad\ Reporter) \setminus TACReporterInterface
\end{array}
$$

I.1.5 Steam Boiler

$\textbf{process}\ \mathit{SteamBoiler} \mathrel{\widehat=} \mathit{TACReporter}$

I.2 Resultant CSP and Z Model

Available either in accompanying CD-ROM or online at GitHub [108].

References

[1] K. Ye and J. Woodcock, "Model checking of state-rich formalism *Circus* by linking to *CSP* $\parallel$ *B*," *International Journal on Software Tools for Technology Transfer*, pp. 1–24, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10009-015-0402-1

[2] M. Weiser, "The Computer for the 21st Century," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999. [Online]. Available: http://doi.acm.org/10.1145/329124.329126

[3] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, Jun. 2010, pp. 731–736.

[4] J. M. Spivey, *Z Notation: A Reference Manual (2. ed.)*, ser. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[5] J.-R. Abrial, *The B-book: assigning programs to meanings.* Cambridge University Press, 2005.

[6] C. B. Jones, *Systematic software development using VDM (2. ed.)*, ser. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[7] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[8] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[9] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92.

[10] J. A. Bergstra and J. W. Klop, "Process Algebra for Synchronous Communication," *Information and Control*, vol. 60, no. 1-3, pp. 109–137, 1984.

[11] C. Fischer, *Formal Methods for Open Object-based Distributed Systems: Volume 2.* Boston, MA: Springer US, 1997, ch. CSP-OZ: A Combination of Object-Z and CSP, pp. 423–438. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-35261-9_29

[12] A. Galloway and B. Stoddart, "An Operational Semantics for ZCCS," in *ICFEM*, 1997, pp. 272–.

[13] A. W. Roscoe, J. Woodcock, and L. Wulf, "Non-interference through Determinism," *Journal of Computer Security*, vol. 4, no. 1, pp. 27–54, 1996.

[14] A. Mota and A. Sampaio, "Model-Checking CSP-Z," in *FASE*, 1998, pp. 205–220.

[15] S. Schneider and H. Treharne, "CSP theorems for communicating B machines," *Formal Asp. Comput.*, vol. 17, no. 4, pp. 390–422, 2005.

[16] C. Fischer, *ZUM '98: The Z Formal Specification Notation: 11th International Conference of Z Users, Berlin, Germany, September 24-26, 1998. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, ch. How to Combine Z with a Process Algebra, pp. 5–23. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-49676-2_2

[17] H. W. Clemens Fischer, "Model-Checking CSP-OZ Specifications with FDR," 1999.

[18] FDR2: a refinement checker for establishing properties of models expressed in CSP. [Online]. Available: www.fsel.com/software.html

[19] A. Mota and A. Sampaio, "Model-checking CSP-Z: strategy, tool support and industrial application," *Sci. Comput. Program.*, vol. 40, no. 1, pp. 59–96, 2001.

[20] The ProB Animator and Model Checker. [Online]. Available: http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page

[21] J. Woodcock and A. Cavalcanti, "The Semantics of Circus," in *ZB 2002:Formal Specification and Development in Z and B.* Springer, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1007/3-540-45648-1_10

[22] C. C. Morgan, *Programming from specifications, 2nd Edition*, ser. Prentice Hall International series in computer science. Prentice Hall, 1994.

[23] J. Woodcock and A. Cavalcanti, "A Concurrent Language for Refinement," in *5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, 16-17 July 2001*, 2001. [Online]. Available: http://ewic.bcs.org/content/ConWebDoc/4146

[24] C. Hoare and J. He, *Unifying theories of programming.* Prentice Hall, 1998, vol. 14.

[25] K. Wei, J. Woodcock, and A. Burns, "Timed Circus: Timed CSP with the Miracle," in *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, April 2011, pp. 55–64.

[26] A. Cavalcanti, A. Sampaio, and J. Woodcock, "Unifying classes and processes," *Software & Systems Modeling*, vol. 4, no. 3, pp. 277–296, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10270-005-0085-2

[27] E. W. Dijkstra, *A Discipline of Programming.* Prentice-Hall, 1976.

[28] G. Smith, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods: 4th International Symposium of Formal Methods Europe Graz, Austria, September 15–19, 1997 Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ch. A semantic integration of object-Z and CSP for the specification of concurrent systems, pp. 62–81. [Online]. Available: http://dx.doi.org/10.1007/3-540-63533-5_4

[29] D. A. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith, "Object-Z: An Object-Oriented Extension to Z," in *Formal Description Techniques II, FORTE'89*, S. T. Vuong, Ed. North-Holland, 1990, pp. 281–296.

[30] D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science).* sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.

[31] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking.* Cambridge, MA, USA: MIT Press, 1999.

[32] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the State Explosion Problem in Model Checking," in *Informatics*. Springer, Jan. 2001. [Online]. Available: http://dx.doi.org/10.1007/3-540-44577-3_12

[33] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. Model Checking and the State Explosion Problem, pp. 1–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35746-6_1

[34] M. Oliveira, A. Cavalcanti, and J. Woodcock, "Unifying Theories in ProofPower-Z," in *Unifying Theories of Programming*. Springer, Jan. 2006. [Online]. Available: http://dx.doi.org/10.1007/11768173_8

[35] M. V. Oliveira, "Formal Derivation of State-Rich Reactive Programs using Circus," Ph.D. dissertation, University of York, 2005.

[36] M. Oliveira, A. Cavalcanti, and J. Woodcock, "Unifying theories in ProofPower-Z," *Formal Aspects of Computing*, vol. 25, no. 1, p. 133, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1007/s00165-007-0044-5

[37] ProofPower. [Online]. Available: http://www.lemma-one.com/ProofPower/index/index.html

[38] F. Zeyda and A. Cavalcanti, "Mechanical Reasoning About Families of UTP Theories," *Electron. Notes Theor. Comput. Sci.*, vol. 240, pp. 239–257, Jul. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2010.02.010

[39] ——, "Encoding Circus Programs in ProofPowerZ," in *Unifying Theories of Programming*. Springer, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14521-6_13

[40] A. Feliachi, M.-C. Gaudel, and B. Wolff, "Isabelle/Circus: A Process Specification and Verification Environment," in *VSTTE*, 2012, pp. 243–260.

[41] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[42] S. Foster and J. Woodcock, "Unifying Theories of Programming in Isabelle," in *Unifying Theories of Programming and Formal Engineering Methods*. Springer, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39721-9_3

[43] S. Foster, F. Zeyda, and J. Woodcock, "Isabelle/UTP: A Mechanised Theory Engineering Framework," *Unifying Theories of Programming*, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-14806-9_2

[44] L. Freitas, "Model checking *circus*," Ph.D. dissertation, Department of Computer Science, The University of York, UK, 2005, yCST-2005/11.

[45] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[46] M. Saaltink, *The Z/EVES system*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 72–85. [Online]. Available: http://dx.doi.org/10.1007/BFb0027284

[47] A. Mota, A. Farias, A. Didier, and J. Woodcock, "Rapid Prototyping of a Semantically Well Founded Circus Model Checker," *Software Engineering and Formal Methods*, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10431-7_17

[48] A. Mota, A. Farias, J. Woodcock, and P. G. Larsen, "Model checking CML: tool development and industrial applications," *Formal Aspects of Computing*, vol. 27, no. 5-6, p. 975, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1007/s00165-015-0342-2

[49] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Reasoning about Metamodeling with Formal Specifications and Automatic Proofs," *Model Driven Engineering Languages and Systems*, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24485-8_48

[50] A. Freitas and A. Cavalcanti, "Automatic Translation from Circus to Java," in *FM*, 2006, pp. 115–130.

[51] M. Oliveira, A. Cavalcanti, and J. Woodcock, "Formal Development of Industrial-Scale Systems in Circus," *ISSE*, vol. 1, no. 2, pp. 125–146, 2005.

[52] A. Beg and A. Butterfield, "Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation," in *FIT*, 2010, p. 47.

[53] ——, "Development of a prototype translator from Circus to CSPm," in *2015 International Conference on Open Source Systems Technologies (ICOSST)*, Dec 2015, pp. 16–23.

[54] M. Oliveira, A. Sampaio, P. Antonino, R. Ramos, A. Cavalcanti, and J. Woodcock, "Compositional Analysis and Design of CML Models," COMPASS Deliverable D24.1, March 2013.

[55] M. Oliveira, A. Sampaio, and M. Conserva Filho, "Model-Checking Circus State-Rich Specifications," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, E. Albert and E. Sekerinski, Eds. Springer International Publishing, 2014, pp. 39–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-10181-1_3

[56] S. Barrocas and M. Oliveira, "JCircus 2.0: an Extension of an Automatic Translator from Circus to Java," in *Communicating Process Architectures 2012*, P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen, and A. T. Sampson, Eds., aug 2012, pp. 15–36.

[57] P. H. Welch, "Process Oriented Design for Java: Concurrency for All," in *PDPTA*, 2000.

[58] B. Scattergood, "The Semantics and Implementation of Machine-Readable CSP," Ph.D. dissertation, University of Oxford, 1998.

[59] S. Nogueira, A. Sampaio, and A. Mota, "Test generation from state based use case models," *Formal Aspects of Computing*, pp. 1–50, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00165-012-0258-z

[60] *FDR2 User Manual*, Fdr 2.94 ed., Formal Systems (Europe) Ltd, May 2012.

[61] J. C. P. Woodcock, A. L. C. Cavalcanti, J. Fitzgerald, P. G. Larsen, A. Miyazawa, and S. Perry, "Features of CML: a Formal Modelling Language for Systems of Systems," in *7th International Conference on System of System Engineering*, ser. IEEE Systems Journal, vol. 6. IEEE, 2012.

[62] P. Malik and M. Utting, “CZT: A Framework for Z Tools,” in *ZB*, 2005, pp. 65–84.

[63] J. Woodcock and J. Davies, *Using Z: specification, refinement, and proof.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[64] A. Cavalcanti, A. Sampaio, and J. Woodcock, “A Refinement Strategy for Circus,” *Formal Asp. Comput.*, vol. 15, no. 2-3, pp. 146–181, 2003.

[65] J.-R. Abrial, “Steam-Boiler Control Specification Problem,” in *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995).*, 1995, pp. 500–509. [Online]. Available: http://dx.doi.org/10.1007/BFb0027252

[66] J. Woodcock, “A Circus Steam Boiler: using the unifying theory of Z and CSP,” Oxford University Computing Laborator, Tech. Rep., 2001.

[67] J. Woodcock and A. Cavalcanti, “The Steam Boiler in a Unified Theory of Z and CSP,” in *8th Asia-Pacific Software Engineering Conference (APSEC 2001), 4-7 December 2001, Macau, China*, 2001, pp. 291–298. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2001.991490

[68] A. Cavalcanti and J. Woodcock, *Refinement Techniques in Software Engineering: First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004 Revised Lectures.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. A Tutorial Introduction to CSP in Unifying Theories of Programming, pp. 220–268. [Online]. Available: http://dx.doi.org/10.1007/11889229_6

[69] J. Woodcock and A. Cavalcanti, *Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ch. A Tutorial Introduction to Designs in Unifying Theories of Programming, pp. 40–66. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24756-2_4

[70] *ISO/IEC: Information Technology-Z Formal Specification Notation-Syntax, Type System and Semantics.*, International Standard Std. 13 568, 2002. [Online]. Available: http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip

[71] A. Roscoe, *Understanding Concurrent Systems*, 1st ed. New York, NY, USA: Springer-Verlag New York, Inc., 2010.

[72] M. Oliveira, A. Cavalcanti, and J. Woodcock, “A Denotational Semantics for Circus,” *Electr. Notes Theor. Comput. Sci.*, vol. 187, pp. 107–123, 2007.

[73] ——, “A UTP Semantics for Circus,” *Formal Asp. Comput.*, vol. 21, no. 1-2, pp. 3–32, 2009.

[74] J. Woodcock, A. Cavalcanti, and L. Freitas, *FM 2005: Formal Methods: International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, ch. Operational Semantics for Model Checking Circus, pp. 237–252. [Online]. Available: http://dx.doi.org/10.1007/11526841_17

[75] J. Woodcock, A. Cavalcanti, M.-C. Gaudel, and L. Freitas, “Operational Semantics for Circus,” *Formal Aspects of Computing*, 2007. [Online]. Available: https://www.cs.york.ac.uk/ftpdir/pub/leo/utp/journal-pub/circus-operational-semantics.pdf

[76] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60, no. 61, pp. 17–139, 2004.

[77] M. J. Butler, "csp2B: A Practical Approach to Combining CSP and B," *Formal Asp. Comput.*, vol. 12, no. 3, pp. 182–198, 2000.

[78] C. Morgan, *Of wp and CSP.* New York, NY: Springer New York, 1990, pp. 319–326. [Online]. Available: http://dx.doi.org/10.1007/978-1-4612-4476-9_37

[79] M. Butler and M. Leuschel, "Combining CSP and B for Specification and Property Verification," pp. 221–236, 2005.

[80] T. Miller, L. Freitas, P. Malik, and M. Utting, "CZT Support for Z Extensions," in *IFM*, 2005, pp. 227–245.

[81] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[82] M. Oliveira, A. C. Gurgel, and C. G. de Castro, "CRefine: Support for the Circus Refinement Calculus," in *SEFM*, 2008, pp. 281–290.

[83] M. Carlsson, *Sicstus PROLOG User's Manual 4.2.* Books On Demand - Proquest, 2012.

[84] D. Plagge and M. Leuschel, "Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more," *STTT*, vol. 12, no. 1, pp. 9–21, 2010.

[85] ——, *Integrated Formal Methods: 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ch. Validating Z Specifications Using the ProB Animator and Model Checker, pp. 480–500. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73210-5_25

[86] Clearsy. B LANGUAGE REFERENCE MANUAL (Version 1.8.6). [Online]. Available: http://www.atelierb.eu/ressources/manrefb1.8.6.uk.pdf

[87] CADiZ. [Online]. Available: http://www.cs.york.ac.uk/hise/cadiz/

[88] M. L. Daniel Plagge, "Validating Z Specifications using the ProB Animator and Model Checker," *Integrated Formal Methods: 6th International Conference, IFM 2007*, 2007.

[89] A. Cavalcanti and J. Woodcock, "ZRC – A Refinement Calculus for Z," *Formal Aspects of Computing*, vol. 10, no. 3, pp. 267–289, 1998. [Online]. Available: http://dx.doi.org/10.1007/s001650050016

[90] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic," in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '86. New York, NY, USA: ACM, 1986, pp. 184–193. [Online]. Available: http://doi.acm.org/10.1145/512644.512661

[91] R. Lazic, "A Semantic Study of Data Independence with Application to Model Checking," Ph.D. dissertation, 1999. [Online]. Available: http://citeseer.uark.edu:8080/citeseerx/showciting;jsessionid=344C8C9F9CE25F2A4FE75DDD6362C072?cid=96955&#38;sort=recent

[92] A. Farias, A. C. Mota, and A. Sampaio, "Compositional Abstraction of CSPZ Processes," *J. Braz. Comp. Soc.*, vol. 14, no. 2, pp. 23–44, 2008. [Online]. Available: http://dx.doi.org/10.1590/S0104-65002008000200003

[93] SICStus Prolog Manual (Arithmetic Expressions). [Online]. Available: https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/ref_002dari_002daex.html#ref_002dari_002daex

[94] D. Leijen, "Division and Modulus for Computer Scientists," 2001.

[95] M. Leuschel and M. J. Butler, "Automatic Refinement Checking for B," in *ICFEM*, 2005, pp. 345–359.

[96] J. C. Bauer, *Specification for a software program for a boiler water content monitor and control system.* University of Waterloo, Institute of Risk Research, 1993.

[97] J.-R. Abrial, E. Börger, and H. Langmaack, "The Stream Boiler Case Study: Competition of Formal Program Specification and Development Methods," in *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995).*, 1995, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1007/BFb0027228

[98] J.-R. Abrial, E. Börger, and H. Langmaack, Eds., *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, ser. Lecture Notes in Computer Science, vol. 1165. Springer, 1996.

[99] M. Leuschel, M. J. Butler, C. Spermann, and E. Turner, "Symmetry Reduction for B by Permutation Flooding," in *B*, 2007, pp. 79–93.

[100] I. Dobrikov and M. Leuschel, "Optimising the ProB model checker for B using partial order reduction," *Formal Aspects of Computing*, p. 1, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1007/s00165-015-0351-1

[101] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[102] Y. Yu, P. Manolios, and L. Lamport, *Model Checking TLA+ Specifications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. [Online]. Available: http://dx.doi.org/10.1007/3-540-48153-2_6

[103] S. Nogueira, A. Sampaio, and A. Mota, "Test generation from state based use case models," *Formal Aspects of Computing*, vol. 26, no. 3, pp. 441–490, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00165-012-0258-z

[104] K. Ye, "CSP Libraries for implementation of Circus Operators," 2016. [Online]. Available: https://github.com/RandallYe/Circus-Programs/tree/master/Cases/Common/csp_libs

[105] ——, "Circus2ZCSP: A translator from Circus to CSP || Z in terms of CSP || B." [Online]. Available: https://github.com/RandallYe/Circus2ZCSP

[106] ——, "Source Models for Bounded Reactive Buffer Case Study," 2016. [Online]. Available: https://github.com/RandallYe/Circus-Programs/tree/master/Cases/ReactiveBuffer

[107] ——, "Source Models for ESEL Case Study," 2016. [Online]. Available: https://github.com/RandallYe/Circus-Programs/tree/master/Cases/ESEL

[108] ——, "Source Models for Steam Boiler Case Study," 2016. [Online]. Available: https://github.com/RandallYe/Circus-Programs/tree/master/Cases/SteamBoiler

Index