Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Model-Checking *Circus* with FDR using *Circus2CSP*

Artur Oliveira Gomes

School of Computer Science and Statistics
Trinity College Dublin
Dublin, Ireland

A thesis submitted for the degree of

Doctor of Philosophy

2019

To four women, four loves.

My grandmother, Ivonise,

My mother, Vilma,

My beloved wife, Lidiane,

And my daughter, Maria Sofia.

Acknowledgements

Being a student is not an easy role. Building our knowledge is constant throughout most of our lives. It requires dedication, attention, motivation and passion. I consider myself as a very lucky person for having the opportunity for studying until reaching the highest degree. Coming from a country where education is still far from being a priority for the government, I am really grateful for being one of the chosen ones to have the opportunity to study abroad funded by my own country.

However, none of this would be possible if my education didn't start at home. I will always be grateful for my parents, for all the efforts to help me be the person I am now. Thank you, dad, for inspiring me to become a lecture like you. I feel myself in the same position you were twenty years ago when you submitted your thesis. And Mom, I know you are proud of me wherever you are now. Saying goodbye to you was the hardest thing I've ever done, but I know you're in a better place now. All your effort throughout all these years helping me succeed will always be in my memories.

Even if I could, I would never find enough words to describe my deepest gratitude to my beloved wife, Lidiane, for holding my hand so tight that I was able to reach the end of this doctorate. Her presence in my life was essential for helping me keep my sanity after all the difficult moments we've been together. She always held me with kindness and love. Together, we were gifted with our lovely daughter, Sofia, our daily portion of joy and happiness, even in the cold and days of winter, giving us warm hugs and smiles.

To my grandmother Ivonise, my gratitude for her kind words of motivation through the phone, even knowing all her suffering in her 90 years.

To my supervisor, Dr Andrew Butterfield, I can't be grateful enough for supervising me throughout the past four years. His insights and advices went far beyond the technical aspects of my research. His understanding of my difficulties along with his uncountable advices helped me to grow not only as a researcher but also as a man.

To my psychotherapists, Trish Murphy and Dr Niamh Farrely, my deepest gratitude for all the support provided helping me keep my progress through the difficult times.

I'd like to thank my working colleagues from the Campus of Pantanal - the Federal University of Mato Grosso do Sul, for supporting me during my leave for studies.

For the suggestions and comments that helped me through my research, I'd like to thank Prof. Jim Woodcock and Dr Simon Foster. To my friend Samuel Barrocas, a big thank you for sharing his experiences with *JCircus*.

Finally, Dublin was home for my family and me for four years. I met nice people and made nice friends. I'd like to acknowledge my gratitude to my friends from our office in Trinity College: Aimee Borda, Marian Reeves, Daniel Flynn, among others. We shared so many good moments where we

could discuss the most random philosophical facts of the daily life, always resulting in good laughs. Thanks also to the students, researchers and lecturers that joined us in our coffee club, when we used to leave our tasks in our offices and have a break with friendly conversations with a nice cup of coffee. To my friends in Brazil, thank you for your support, even being physically distant.

This work was funded by CNPq (Brazilian National Council for Scientific and Technological Development) within the Science without Borders programme, Grant No. 201857/2014-6, and partially funded by Science Foundation Ireland grant 13/RC/2094.

Model-Checking *Circus* with FDR using *Circus2CSP*

Artur Oliveira Gomes

Abstract

The current lack of tool support for model-checking *Circus*, a formalism which combines Z, CSP, refinement calculus and Dijkstra's guarded commands, is one of the constraints for its use at industrial scale. Nowadays, it is possible to translate *Circus* to other formalisms and to verify them profiting from existing model-checkers, such as FDR and ProB, which have no direct support for *Circus*. However, such approaches are usually performed manually, which requires time and also may introduce errors. We used Haskell to implement an automatic tool which preserves the semantics while translating *Circus* to CSP, which includes an automatic *Circus* refinement calculator as part of the transformation before the translation into CSP. It is based on the existing set of translation rules but has several improvements, compared to its original strategy. We extended the language coverage, and provide a more efficient structure for handling more complex type systems, and better support for specifications with larger sets of state variables. We introduce a set of translation rules for using Z schemas as *Circus* actions, not previously supported in the translation into CSP. In our research, we also explored ways of verifying the correctness of our implementation. We proved manually some interesting properties for the translation, which are related to the improvements developed in this work. We also investigated the use of Isabelle/UTP as a theorem prover in order to help us to increase confidence in our approach. However, our tool was tested using several *Circus* case-studies present in the literature. Among our findings, we observed that, compared to previous work, the state-space explored with our improved translation strategy was reduced dramatically by over eighty per cent and the time consumed for checks in FDR was reduced from minutes to milliseconds, compared to the initially adopted strategy.

Declaration

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Artur Oliveira Gomes

Contents

1 Introduction **1**

1.1 Thesis outline . . . 5

2 Literature Review **7**

2.1 Formal Languages and Tools . . . 7

2.1.1 State-based languages . . . 7

2.1.2 Process-based languages . . . 8

2.1.3 Combining State-based languages with Process-based languages . . . 8

2.2 Model-Checking . . . 9

2.3 Refinement of Specifications . . . 13

2.4 Quick *Circus* Guide . . . 15

2.4.1 *Circus* Background . . . 15

2.4.2 *Circus* through an example . . . 17

2.5 Thesis Proposition . . . 18

3 Methodology **20**

3.1 Translating *Circus* to CSP_M . . . 20

3.2 The Memory Model . . . 22

3.3 Extending Jaza . . . 23

3.3.1 Jaza supporting *Circus* . . . 24

3.4 Implementing a *Circus* Refinement Laws Calculator . . . 26

3.4.1 A simple refinement example . . . 29

3.5 Rewriting Z Schemas into *Circus* Actions . . . 30

3.5.1 An Example of the Transformation . . . 35

3.5.2 Final considerations . . . 37

4 Rewritting *Circus* state-rich processes with Ω **38**

4.1 Upgrading the Memory Model . . . 41

4.1.1 Limitation 1: Z types vs. CSP_M types . . . 41

4.1.2 Limitation 2: FDR time/space explosion . . . 42
4.2 Rewritting *Circus* Actions with Ω . . . 43

5 Translating *Circus* to CSP_M with Υ 48
5.1 The CSP_M type environment for the *Memory* model . . . 48
5.2 Generating Bindings . . . 50
5.3 Mapping *Circus* Processes - Υ_P . . . 51
5.4 Mapping *Circus* Actions - Υ_A . . . 51
5.5 The CSP_M version of the *WakeUp* process . . . 52

6 A quick overview of *Circus2CSP* 54
6.1 Commands for the translation to CSP_M . . . 54
6.2 Injecting CSP_M in the Source Files . . . 55
6.3 Integrating FDR4 with *Circus2CSP* . . . 55
6.4 Outputs provided by *Circus2CSP* . . . 56
6.5 Final Considerations . . . 57

7 Using *Circus2CSP* and FDR4 58
7.1 Simple *Circus* Examples . . . 58
7.2 Evaluating *Circus2CSP* with Case-Studies . . . 65
7.3 The Haemodialysis Case Study . . . 65
7.3.1 The Haemodialysis Machine in *Circus* . . . 66
7.3.2 System Requirements . . . 67
7.3.3 Translating it into CSP_M . . . 67
7.3.4 HD Machine Experiments . . . 70
7.4 Ring-Buffer Experiments . . . 72
7.5 Compression Experiments . . . 74
7.6 Final considerations for testing *Circus2CSP* . . . 75

8 Validating the Translation 76
8.1 Verifying the Refinements on the Memory Model . . . 76
8.1.1 Memory Model Refinement Proof . . . 76
8.1.2 Memory Model Refinement using FDR . . . 78
8.2 Plans for the Verified Translation . . . 79

9 Conclusions 82
9.1 Contributions . . . 82

9.2 Future Work . . . 84

A *Circus* AST **100**

A.1 *Circus* and Z Paragraphs . . . 100

A.2 *Circus* Channel Declaration – *CDecl* . . . 100

A.3 *Circus* Process – *ProcDecl* . . . 100

A.4 *Circus* Process – *ProcessDef* . . . 100

A.5 *Circus* Process – *CProc* . . . 101

A.6 Process paragraphs – *PPar* . . . 101

A.7 Parametrised Actions – *ParAction* . . . 101

A.8 *Circus* Actions – *CAction* . . . 102

A.9 *Circus* Communication . . . 102

A.10 *Circus* Communication – *CParameter* . . . 102

A.11 *Circus* Commands – *CCommand* . . . 103

A.12 *Circus* Guards – *CGActions* . . . 103

A.13 *Circus* Renaming – *CReplace* . . . 103

B Omega Mapping Functions **104**

B.1 *Circus* and Z Paragraphs . . . 104

B.2 Mapping *Circus* Processes Declaration . . . 104

B.2.1 *Circus* Process – *ProcDecl* . . . 104

B.2.2 *Circus* Process – *ProcessDef* . . . 104

B.2.3 *Circus* Process – *CProc* . . . 105

B.3 Mapping Parameterised *Circus* Actions . . . 105

B.3.1 Parametrised Actions – *ParAction* . . . 105

B.3.2 Stateless *Circus* - Actions - Derived from Deliverable 24.1 [132] . . . 105

B.3.3 Definitions of Ω'_A . . . 108

B.4 Γ functions . . . 110

B.4.1 Stateless *Circus* - Actions . . . 110

B.4.2 Definitions of Γ'_A . . . 112

C Mapping Functions - *Circus* to CSP **115**

C.1 Mapping *Circus* Paragraphs . . . 115

C.1.1 Mapping *Circus* Channels . . . 116

C.1.2 Mapping Function for Channel Set Expressions . . . 116

C.1.3 Mapping *Circus* Processes Declaration . . . 116

C.1.4 Mapping *Circus* Processes Definition . . . 116
C.1.5 Mapping *Circus* Processes . . . 117
C.1.6 Mapping *Circus* Processes Paragraphs . . . 117
C.1.7 Mapping *Circus* Actions . . . 118
C.1.8 Mapping *Circus* Commands . . . 118
C.1.9 Mapping *Circus* Guarded Actions . . . 118
C.2 Mapping Functions from *Circus* to CSP - Based on D24.1 - COMPASS . . . 118
C.2.1 Mapping Function for Expressions . . . 119
C.2.2 Mapping Functions for Predicates . . . 120

D *Circus* refinement laws **121**
D.1 Lemmas from Deliverable 24.1 [132] . . . 124
D.2 *Circus* Denotational Semantics [129, Chapter 3] . . . 124

E Beyond the Tool Implementation - Linking with Isabelle/UTP **125**

F From Haskell to Isabelle using Haskabelle **126**
F.1 Reasoning using Isabelle/UTP . . . 127
F.1.1 Considerations for Future Work . . . 129

G Proofs on *Circus* models **131**
G.1 Proofs on the Z schemas refinement - *Chronometer* . . . 131
G.2 Assignment then Choice problem . . . 132
G.2.1 Assignment then choice - Both sides . . . 132
G.2.2 Assignment then choice - Assignment in one side . . . 142

H Chronometer models **151**
H.1 Original models . . . 151
H.2 After the Z Schemas Translation . . . 153
H.3 CSP_M code from *Circus2CSP* . . . 154
H.4 *WakeUp* . . . 157
H.4.1 *WakeUp* using *Circus2CSP* . . . 158

List of Figures

2.1 Strategies for translating *Circus* into CSP. 19

3.1 Structure of *Circus2CSP* 20

3.2 Mapping *Circus* into CSP_M (derived from [132, Fig.7, p77]) 21

3.3 Jaza loading LaTeX files. 25

3.4 Experiment results derived from Table 3.1 - $D241$, $D241Inv$, $CTOC$ and $CTOCPre$. . . 34

3.5 Experiment results derived from Table 3.1 - $D241InvPre$, $D241Pre$, $CTOC$ and $CTOCPre$. 35

3.6 Experimenting real chronometer values (0 up to 60) derived from Table 3.1 35

4.1 Converging transitions after the state initialisation. 41

6.1 *Circus2CSP* initial screen 54

6.2 *Circus2CSP* help menu 55

6.3 *Circus2CSP* help menu - continuation of Fig. 6.2 56

7.1 Execution flow of the Haemodialysis machine 66

8.1 Plans for verifying the correctness of *Circus2CSP* 79

F.1 Mapping Haskell to Isabelle/UTP 128

F.2 Refinement test 130

List of Tables

3.1 Interference of invariants and preconditions in CSP_M- Deadlock freedom checks 34

7.1 Time for asserting deadlock freedom of the HD Machine in FDR4 72

7.2 *RingBuffer* checks: deadlock and livelock freedom, and determinism. 73

7.3 Refinement checks between three models of the Ring Buffer example 73

7.4 Experimenting CSP_M compression techniques with the HD Machine 74

Chapter 1

Introduction

The constant growth of technology associated with computing and the need for increasingly secure software has increased the demand for process verification systems, especially in the area of formal methods. Such research field aims to apply a set of methodologies and formal languages to rigorously specify, develop, and verify systems both in hardware and in software.

The application of the methodologies developed in the area of formal methods achieves a greater depth in the analysis of computer systems that would be possible otherwise. Currently, formal methods and its tools have reached such a level of usability that allows its application on an industrial scale [10, 17, 26, 31, 11], allowing software developers to provide more meaningful guarantees for their projects.

Researchers in this area have been seeking solutions to ensure the integrity of systems, in particular, critical systems. When we speak of critical systems, we refer to those with low failure tolerance or even intolerant to failures. These are systems that, in the case of failures, often involve risk to human beings, such as sensors and autopilots in avionics, and for medical devices, such as pacemakers and defibrillators.

Related to avionics, we report on the DO-178B document, *Software Considerations in Airborne Systems and Equipment Certification* [144] produced by RTCA, along with the *European Organisation for Civil Aviation Equipment* (EUROCAE), whose objective is to define criteria for certification of airborne systems, intended for software development teams for the aviation industry.

According to this document, one of the goals of the verification process is to ensure that the resulting software complies with its requirements. The document specifies the classification of software at various levels of criticality, according to the effects of a fault condition. It starts from levels at which there is no impact (no consequences) or which have minimal impact on the system to those in which result in catastrophic failures to the aircraft, jeopardising the safety of flight, such as a fall or mid-air collision.

Due to the complexity and the safety criteria of such a system, it is necessary to adopt a consistent methodology for specification, modelling, verification and validation, comprehensively, aiming to meet the highest standards of certification of software. Twenty years later, the DO-178C [65] was released and includes in its standards the use of formal methods as a way to overcome the difficulties presented above.

There is also an interest in applying formal methods while developing medical devices. It is well known that such devices deal directly with health, acting in many ways, for the preservation of life, so that the security of these devices is of utmost importance. As for the avionics software systems, there are also agencies responsible for regulating the production of medical devices, such as in the United States, where the *Food and Drug Administration* (FDA) controls the production of such systems in that country. In the last five years, several research groups in the area of formal methods, scattered around the world, have been working on a case study whose goal is the specification, verification and prototyping of a cardiac pacemaker [23].

For its relevance, this study is an excellent example of research of a critical system, driven by the assumption that it is necessary to have a firm assurance that such a system will work according to its requirements [24]. It is known that any failure or error in the system can cause serious health problems in patients who are dependent on a pacemaker to survive.

Related to the pacemaker challenge, much work has been done using various approaches. For instance, languages like Z [78, 77], *Vienna Development Model*, VDM [18, 107], Event-B [113, 154, 153], *abstract state machine* (ASM) [12], and CSP[149], have been addopted in order to formalise the pacemaker system. Moreover, a range of analysis has been performed using techniques and tools such as model refinement [155, 114], validation using ProB [135] and model checking [158, 38]. Moreover, test cases have been created [94, 115, 112].

More recently, the ABZ Conference has proposed the adoption of an industrial scale case-study so participants can formalise the specification and submit to the conference. That is a very positive way of comparing formal languages and tools among several approaches to formalise the case-study. A first example was proposed for ABZ 2014, with the Landing Gear [21] Case Study. For the ABZ 2016, The Haemodialysis Machine [7] Case Study was proposed.

The use of formal methods provides a way to specify, develop, and verify complex systems rigorously, used in order to automate various kinds of verification steps. Among several approaches aiming at the correctness of systems, model-checking formally assesses given systems regarding their desired/undesired behavioural properties, through exhaustive checking of a finite model. Our intention in undertaking the Haemodialysis case-study was to apply our knowledge in a real-life example in order to identify the limitations in our approach regarding the formal language and tool support as well.

We decided to use *Circus* [75] as a formal language based on our previous experience with the language and explore its capabilities in order to model the haemodialysis system. We also wanted to explore how we could analyse the system using existing tools and how we could format our model in order to be supported and analysed by such tools.

Currently, there is a limited tool support for model-checking *Circus*. In order to overcome such problem, one can translate *Circus* into CSP by hand, and then, model-check the translated model using FDR, which supports programs written in CSP. It analyses failures and divergence models, with checks related to, for example, deadlock, livelock and termination. However, we have to somehow translate *Circus* into CSP, by adapting the model to the CSP restrictions. For instance, we have to capture the state-based features of *Circus* in CSP.

Until now, tools for translating *Circus* into CSP are limited. There have been a few related works in defining a translation strategy to obtain CSP programs from *Circus*. Ye and Woodcock [175], defined a link from *Circus* to $CSP \| B$ with model-checking using ProB [135]. Moreover, Beg [16], also prototyped a tool for translating *Circus* to CSP_M. These two are further discussed in the next section, but so far, their current version are limited to a restricted subset of *Circus*.

As an initial attempt to model-check *Circus*, we participated in the ABZ'16 haemodialysis case study [75], producing a *Circus* specification, manually translating it into CSP_M, which we then checked with FDR[72]. Moreover, when translating *Circus* into CSP, we adapted the *Circus* model to map the structural Z parts into appropriate CSP.

Unlike in *Circus* processes, an explicit notion of state variables is not present in CSP processes. Therefore, in order to translate *Circus* state, we would either translate it into a memory process [122, 86, 140], allowing other processes to read and write the values by synchronising on memory 'get' and 'put' events, or to transform the state variables into process parameters, as used by Beg [16]. For instance, we captured the state-based features of *Circus* in CSP using a memory process synchronising on channels for reading and updating the values of the state variables. Such an approach was also used while model checking [74] the ARINC 653 [4] architecture.

Our formalised model of the haemodialysis case-study is presented in Section 7.3. We describe the decisions taken in order to structure the system, such as how we capture the system data using Z schemas as a *Circus* state and how *Circus* processes and actions capture the behaviour of the system, and finally, how *Circus* channels are used to model the communication between the components of the system.

The handmade translation uses a significant but rather simple structure of 'gets' and 'sets' functions and channels for reading and updating the values of the state variables in the process. Such an approach requires the creation of two channels for each state variable, which returns and updates the value of a variable, over the state, which in its turn, is defined as a tuple.

The translation of that case study has shown to be already hard to perform since we had doubled the number of lines of the specification, from a thousand (*Circus* model) to over two thousand (CSP-translated model). Our research showed that this handmade approach can be error-prone, and is unlikely to be feasible for industrial scale systems. Therefore, an automatic tool would be useful for this task.

In light of these findings, we present here the development of the *Circus2CSP*[1], an automatic tool for model checking *Circus*, where we use a strategy for translating *Circus* programs into CSP_M, aiming at model-checking with FDR. Our first thought was to use our handmade translation strategy. However, our goal was not only to automate but also to produce a smaller and more efficient model compared with existing approaches such as Oliveira's [132], which still captures the intended semantics of a *Circus* specification. Our tool was then built based on the strategy developed during the COMPASS project[2], as presented in the Deliverable 24.1 [132, Section 5.3], that defines a rigorous but manual translation strategy aiming at obtaining CSP_M specifications from *Circus*.

[1]See `https://bitbucket.org/circusmodelcheck/circus2csp`

[2]COMPASS Project - `http://www.compass-research.eu`

During the implementation of our translation tool, we identified the need to improve the translation rules, in particular, for the ones that translate the CSP subset of *Circus* into CSP_M. Our contribution here is not only to implement the tool but to analyse and improve, when required, the current translation strategy and provide new rules, as we present in this thesis. Moreover, we also extend the work of Oliveira *et al.* [132] with an improved definition of the type system approach used in the specification of a memory process in CSP_M, which handles FDR restrictions regarding polymorphic functions.

In summary, our objectives towards model checking *Circus* resulted in the following contributions:

- **Contribution 1.1** *A tool for automatically translating a subset of Circus into CSP_M:*

  Implementation of a tool based on the work of Oliveira et al. [132] where one is able to translate Circus models written in LaTeX into CSP_M, and then, be able to perform model-checking and refinement checks using FDR.

- **Contribution 1.2** *An automatic Circus refinement calculator:*

  As part of the translation strategy, the Circus refinement laws are applied to the processes and actions. In order to automate the translation as much as possible, we provide an automatic Circus refinement calculator.

- **Contribution 1.3** *A transformation of some Z schemas into appropriate Circus constructs for translating into CSP_M:*

  The translation approach presented by Oliveira does not handle Z schemas directly, but only after normalisation. However, such a translation was not yet formally proved to be correct. We explore ways of translating Z schemas into Circus actions, specifically, those schemas where the translation results in a set of assignments.

- **Contribution 1.4** *An improved Circus model that supports multiple types within a specification:*

  The generated CSP_M model from Oliveira et al. using multiple types is not supported by FDR, since it contains some auxiliary functions that are seen by FDR as polymorphic functions, which are not supported by such a tool. We, however, introduce a new data structure that treats each type with its own set of auxiliary functions.

- **Contribution 1.5** *A refinement of the memory model from Oliveira et al. [132]:*

  We provide a refined memory model with distributed memory cells updating and retrieving the values of the state variables, allowing FDR to handle a large number of state variables in a process, optimizing FDR's effort to check such models.

- **Contribution 1.6** *New rules for mapping Circus to CSP_M:*

  We extended the mapping functions for expressions and predicates from Z, as well as mapping functions for those actions specifically related to the Memory model.

- **Contribution 1.7** *A mechanism that integrates Circus2CSP with FDR:*

  We connected our tool to the "terminal-mode" interface of FDR, in order to be able to run checks straight from our tool. Unfortunately, we have no direct access to the code of FDR, and thus, we have to manually parse the results from the execution of FDR's "refine" command.

- **Contribution 1.8** *An automatic assertion generator for checking with FDR:*

  Our tool is able to generate assertion checks for refinement, deadlock, livelock and determinism checks for the loaded specification.

1.1 Thesis outline

In Chapter 2 we start presenting a survey of the state-of-art concerning formal languages and tools, followed by a review of existing work on model checking in Section 2.2. Then, we review refinement approaches in Section 2.3. Furthermore, we introduce *Circus* in Section 2.4. Finally, we conclude the chapter with a discussion of our contributions to the current state-of-art.

We start Chapter 3 with the description of our approach for translating *Circus* to CSP_M, where we detail what kind of model we can generate, as well as implementation decisions. Moreover, in Section 3.5 we introduce an approach for supporting Z schemas in the translation strategy (Contribution 1.3). Finally, an overview of the automatic refinement calculator for *Circus* (Contribution 1.2) is presented in Section 3.4.

Chapter 4 details the transformation of *Circus* processes into a subset that can be then translated into CSP_M. In Section 4.1 we present the improvements for the memory model (Contribution 1.4 and 1.5). Then we present the transformation functions for *Circus* actions in Section 4.2.

Chapter 5 contains the description of how we prepared the CSP_M environment to support the translated model from *Circus*, as well as the decisions taken for the Contribution 1.4 and 1.5, and how these affect the CSP_M code. Moreover, the Contribution 1.6 is described in Section 5.4. Finally, we conclude the chapter with the CSP_M translated version of the *Circus* model presented in Section 2.4.

In Chapter 6, we present our tool *Circus2CSP* (Contribution 1.1), with instructions on how to use it. Then, we introduce how we integrated *Circus2CSP* with FDR (Contribution 1.7) in Section 6.3, along with the instructions for using the assertion generator (Contribution 1.8). Finally, we conclude the chapter with some considerations regarding our experience during the development of the tool.

In Chapter 7 we present our experiments while testing our tool. We also compare our results to the achievements from the literature. In Section 7.3 we illustrate the benefits of using *Circus2CSP* with the example of the Haemodialysis case study, where we compare the results obtained previously [75] with the outcome of using *Circus2CSP*. We discuss experiments using compression techniques in Section 7.5. Finally, we conclude the chapter with some final considerations about our experience while using *Circus2CSP*.

We discuss the proofs on the refinement of the Memory model (Contribution 1.5) in Chapter 8, where a handmade proof is presented, along with an approach for running every proof steps on FDR in order to prove the refinement steps. Moreover, we sketch some plans for using an approach for

validating the implementation as well as the improvements presented in this thesis. In a near future, we intend to produce a link between our tool and Isabelle/UTP, and therefore, verify the Haskell implementation of *Circus2CSP* as well as to integrate our refinement calculator with the theorem prover.

Finally, we conclude this document with Chapter 9 where we discuss how the results obtained throughout our research supports this thesis. Furthermore, provide a summary of our plans for future work.

Chapter 2

Literature Review

A survey of theoretical concepts and related works discussed in this thesis will are presented as illustrated as follows: firstly, we make an analysis of the current state-of-art on formal languages and tools used in the area of formal methods; then, we briefly introduce an overview of model-checking, followed by a survey on refinement strategies. Later in this chapter, we introduce *Circus* followed by a simple example. Finally, we conclude this chapter with some overall conclusions regarding the existing work and the proposition for our thesis.

2.1 Formal Languages and Tools

In this section we survey a few existing formalisms and tools: we first present some state-based languages, such as Z and B; then we detail some process-based languages, like CCS and CSP; then we list a few combinations of these formalisms. In this thesis, we aim at a language and tools that capture both state-based constructs with concurrent processes.

2.1.1 State-based languages

State-based formal languages such as Z [172, 157], B [3] and VDM [62, 5], are used to model structural aspects of a system. By using these languages, we provide a mathematical description of the system, using, for example, set theory, first-order logic and *lambda calculus*. There is, however, a drawback in the use of such model-based languages: modelling aspects of the system behaviour, such as communications between components becomes inconvenient.

The Z language was extended to Object-Z, in order to include notions of object orientation by Carrington*et al.*[32]. The analysis of Z can be specified using theorem provers [106, 145]. Moreover, a refinement calculus for Z, based on the work of Carroll Morgan [121], was introduced by Cavalcanti and Woodcock [34]. Utting [162] developed an animator for Z specifications: a tool written in Haskell that takes Z specifications written in LaTeX and allows the user to interact with the specification. Further work was produced by the Community Z Tools [116], a framework written in Java and aimed at parsing Z specifications and allowing a wide range of assessments. Moreover, Xiaoping [93] presents an approach to produce a Z animator transforming specifications into code written in imperative programming languages.

The language B [3] and Event-B [1] aims at refining specifications to implementations, with tool support [138] such as animator and model-checker (ProB) [102, 99, 101] and the Eclipse-based Rodin toolset [2], which have been successfully used in industrial case-studies [139, 55, 10, 103, 17, 26].

In the context of VDM, the refinement of specifications in such language is possible [125, 18]. Moreover, Couto [47] introduces a proof obligation generator integrated with theorem provers. The use of VDM also extends its support to cover object orientation [63], real-time [163] and distributed systems, using the variants for VDM, VDM++ and VDM-RT. Kanakis [95] presents an extension of the Ouverture [64] tool for the generation of the concurrency aspects of VDM++ models. Finally, code generation for embedded systems is proposed by Bandur *et al.* [13].

2.1.2 Process-based languages

The behavioral aspects of a system can be captured with the help of process-based languages, like *Communicating Sequential Processes* (CSP) [82, 140, 149] and *Calculus of Communicating Systems* (CCS) [117]. For instance, CSP is a language developed to describe and analyse communication and synchronisation between processes in the context of concurrent programs. Roscoe *et al.* [140] presents notions of refinement for CSP and have tool support. Regarding programs written using CSP, we can perform verifications regarding the non-determinism, *deadlock* freedom and *livelock* freedom. Furthermore, the correctness of the software can be analysed with the aid of model analysis tools, via *model-checking* using FDR [72], which also allows the user to animate specifications in CSP. Moreover, Isobe [88] introduced a tool for refinement proofs of CSP. However, unlike the languages based on states, process-based languages does not provide a concise notation for capturing data aspects of systems, becoming inconvenient to define a similar structure for managing the "state" of a system.

2.1.3 Combining State-based languages with Process-based languages

As we aim at the verification of a complex and critical system, we realise that we can not take both structural and behavioural aspects of the system, using the languages presented in this section in isolation. We need a formalisation that combines both aspects, allowing us to create formal models that cover both aspects.

Several formalisms have been combined with the aim of addressing this difficulty. For example, the CSP language and B are integrated by Schneider and Treharne [161], and by Butler and Leuschel [29]. Furthermore, an automatic translation from a combination of B and CSP into Java was presented by Yang and Poppleton [173]. On the other hand, the Z language combined with CSP is addressed in the work of Mota and Sampaio [124] and also is used by Roscoe and Woodcock [143]. Moreover, combinations of Object-Z [32], an extension to the Z language that includes concepts of object orientation, along with CSP, are addressed by several groups [60, 53, 87, 156, 51, 108]. Finally, Galloway and Stoddart adopt the combination of CCS with Z [70] which, in turn, is also adopted by Taguchi and Araki [159].

Woodcock and Cavalcanti defined *Circus* [170], which is a formal language that combines structural aspects of a system using the Z language [172] and the behavioural aspects using CSP [149], along with the refinement calculus [121] and Dijkstra's guarded commands [52]. Its semantics is based

on the *Unifying Theories of Programming (UTP)* [83]. An extension of *Circus* used in order to capture the temporal aspects of systems is presented by Sherif and He, known as *Circus Time* [152, 151], and also by Wei [164].

2.2 Model-Checking

In this work, we are motivated to research techniques for verifying *reactive systems* [148], i.e., those we consider to have an interaction with their environment rather than computing a result in their termination. For instance, the behaviour of a cardiac pacemaker is defined as a set of responses to constantly monitoring the stimuli provided by the human heart, and such a system can be considered as a *reactive system*.

Among the range of verification techniques, model checking is used for exploring all the possible states a reactive system can reach. In other words, the algorithms used for model-checking will evaluate all possible scenarios of a system, which may eventually reach an undesired one, depending on the property that is being evaluated.

The concepts of *model checking* were first introduced almost thirty years ago by Clarke and Emerson [39], and independently and almost simultaneously, by Queille and Sifakis [92], whose goal was to analyse the behaviour of a system model, by given properties, concerning the system formal specifications. The approach proposed by Clarke [40] argues that, rather than using mechanical theorem provers, it would be worth using an algorithm, the *model checker*. It is capable of mechanically asserting if a more concrete model meets a more abstract model, its specification, using propositional temporal logic: it verifies whether a relation $SYS \models f$ (SYS satisfy f), where SYS is a model which can be a *Kripke* structure, of a given formal language, and f is a temporal logic formula.

If the model checker reaches a state where the desired property is violated, it should provide a counter-example [42] indicating how the model has reached such an undesired state. The counter example shows the execution path, with all the steps or transitions, from the initial system state until the state that violates such property under consideration. From that point, with the help of an animator, such as *probe* [105], one can visually repeat such steps, as a way of investigating what led the system to such a state. Therefore, the user can debug its model and fix/adapt it in such a way that the property is no longer evaluated.

It is also fair to mention a few weaknesses of model-checking systems. A first one is related to the system size. Sometimes a system may be larger than the physical memory limits of the computing resource being used. During model-checking, the exhaustive analysis may require to store all the states of the system, whose state space may be exponential in the number of components. In such cases, what happens is the technical challenge called *state explosion* [41]. Some effort has been made in the past decades in order to avoid the state explosion problem, as proposed by McMillan [110], with the use of *symbolic model checking*, which manipulates sets of states represented in propositional logic, instead of single states, using binary decision diagrams (BDD) [27]. Moreover, Biere *et al.* [20] proposed a symbolic model-checking strategy without BDDs, using other boolean decision strategies [22], and using SAT [19] solvers. However, the computational memory available nowadays is much more

accessible than it was a decade ago. Therefore, among the past decades, the size of systems suitable for model-checking went from small examples to real-world applications.

Although there are strategies for avoiding state explosion, as well as the increased computing capacity available nowadays, the problem still occurs, and it is then essential to have in mind the concept of *abstraction* [44]. Usually abstracting systems is an intellectual challenge, and requires some expertise on top of the designer's creativity in order to produce a more abstract model able to cope with the computing limitation. Clarke *et al.* [43] presented a strategy for abstracting models, where a counter example-guided abstraction refinement was proposed. Eventually, should a behaviour occur in the concrete model, that is not present in the abstract model, an erroneous counter example occurs. In such cases, Clarke *et al.*suggested that those counter examples should be used in order to refine the abstraction in order to eliminate such behaviour. Moreover, besides the research on *model abstraction*, compression techniques [142] have been used in model checkers in order to avoid the state space explosion, by reducing the state exploration.

Moreover, the focus of model-checking is on the system's behaviour rather than how the model would manage its data. Therefore, a system whose behaviour strongly relies on its data may become difficult to check, since the data may range over infinite domains. For instance, a small system of a chronometer that has no upper bound limit explicitly defined in the model may force the model-checker to explore an infinite number of states and may never reach an end. Moreover, in some cases where the properties evaluated, such as an atomic proposition such as "volume $\leq$ 200" or "temperature = 100", the checker does not need to assert every possible value of such variables within the range of values of the natural numbers. For the above example, the set of values $\{99, 100, 101, 200, 201\}$ are enough for evaluating the two propositions. Effectively, such optimisation helps the model-checker.

Another weakness of model-checking is related to the completeness of the verification. The checker only verifies the properties provided by the user. However, the lack of completeness comes with the fact that the system properties that are not checked cannot be guaranteed to be correct. Finally, as we are model checking the model of a system, we can not say anything about its final product or prototype, and thus, further investigation such as testing techniques are required.

Our work here focuses more on the implementation of an approach for translating *Circus* into CSP_M, in order to allow the community to model-check CSP_M models derived from *Circus* in FDR. Until now we gave a brief overview on model-checking and will not dive more in-depth into the subject. However, for further reading, an extensive survey on the subject was made by Freitas *et al.* [69]. They details temporal logic and its variations, as well as a more in-depth study on how to avoid state explosion, how to fine tune the formal model using abstractions, and also an analysis between expressiveness and effectiveness when combining model checking with theorem proving.

However, in order to justify our contribution, we provide here a brief survey of the current state of art of approaches for model checking systems designed in the formal languages presented in Section 2.1: we relate works between the mentioned languages with available tools suitable for model-checking.

For the past decades, FDR [71, 72, 73] have been used as a *de-facto* model checker for CSP. Model checking through FDR allows the user to perform a wide range of analysis, such as refinement checks, deadlock and livelock freedom, and termination. Given a CSP process, FDR will convert it

into a *generalised labelled transition system* [136] used to represent such process while checking it for refinement. Briefly, a *labelled transition system* (LTS) represents a system model using a set of states and a collection of transitions between those states. Every transition is *labelled* by an action that happens when the transition occurs, and every state may be *labelled* with a predicate that holds in that state. A *generalised labelled transition system* carries more information than the LTS, where the states are labelled with properties related to the semantic model to be used in the refinement check.

In our work, we use FDR version 4.2, a continuation of the version 3, released in 2013, which was completely redesigned compared to its previous version FDR2. In this new version, a new compiler was designed and proved to be much more efficient than the one used in FDR2, being able to analyse much bigger systems with billions of states. Among the improvements, the new compiler uses a list of strategies to decide how syntactic processes should be compiled. For instance, every CSP operator has a preferred level of compilation, *low-level*, *high-level*, *mixed-level* and *recursive high-level*. The decision of which compilation level will decide which *generalised* LTS representation will be used: **Explicit** or **Super-Combinator**. Those representations differ one to each other in the number of states and time consuming for calculating the transitions. Moreover, the *recursive high-level* strategy is a novelty for FDR3, and the authors argue that such a strategy has reduced the compilation time considerably in many examples [72].

Experiments for comparing FDR3 with other model checkers such as SPIN and DiVinE were conducted [72], aiming at evaluating the checker capability and efficiency concerning memory used. Although in some cases where in-memory checks FDR demonstrated to be slower, it was the only one to be able to compute those checks that required on-disk storage. In summary, the other model checkers evaluated showed great efficiency at the beginning of the checks but became less useful for those checks where the memory was, and disk storage was then required.

It is known that FDR was used in order to model-check some of the combinations of formalisms presented in Section 2.1.3, such as CSP-OZ [61], CSP-Z [56], and Object-Z [96]. Moreover, ProB [102, 99, 103] has been used in order to validate B-Method models, such as the ABZ'14 Landing Gear case-study [100], as well as applied to railway systems [55]. Moreover, Z specifications were animated using ProB [135], using an extension of the tool that supports the Z language.

In addition to FDR, Process Analysis Toolkit (PAT) [158, 54] uses model-checking in order to verify compositional models, complementing FDR checks, including normalisation and simulation. However, it lacks further evaluation in order to compare PAT with FDR. Chaki *et al.* [37] presented a procedure for model checking a combination of state-based with event-based formalisms using the MAGIC (Modular Analysis of proGrams In C) [36] model checker. Finally, the model checker SPIN [84] performs assertion checks using *linear time temporal logic* (LTL) formulae. It differs from FDR, for example, which uses LTS, an approach closely related to the automata theory. Instead, SPIN uses a combination of propositional logic with time-related operators used to formulate complex statements about how a condition can vary over time. Those conditions use past, present and future tense. Usually, LTL is applied to hardware modelling, such as a pipeline circuit, as presented by [28, 19].

Currently, there are limited numbers of tools for verifying *Circus*, such as a refinement calculator, *CRefine* [46], and an animator for *Circus*, Joker [128]. However, for model-checking, *Circus* is usually translated by hand to machine-readable CSP (CSP_M) [146], and then FDR [72] is used. We applied that method in our response to the Haemodialysis case study for ABZ'16 [75]. The approach, however, requires the attention of the researcher, since CSP does not support the state-based specification. It is necessary to produce a modified version of the specification in order to capture the state changes, with the use of communicating channels for updating the state values between processes.

Model-checking *Circus* can be a challenge as the model complexity grows exponentially concerning the state components and actions, for each *Circus* process. Thus, as the number of processes grows, there is a high risk of state-space explosion.

Freitas [68] presented some related work on techniques for model-checking *Circus*, where a refinement model checker based on automata theory [85] and the operational semantics of *Circus* [171] was formalised in Z/Eves [145]. He also prototyped a model checker in Java. It is also known that Mota *et al.* [122, 123] prototypes a strategy for model checking *Circus* using Microsoft FORMULA [90] framework. However, they could not provide a formal proof of the soundness of their approach, since FORMULA does not have an available formal semantics. .

One strategy used in order to overcome the lack of tool support for model checking *Circus* is proposed by Beg [16], who proposes the use of the current CZT [49] framework in order to produce a tool for translating *Circus* into CSP_M. The intention is to build on the existing work from Freitas and Cavalcanti [67], who defined a translation from *Circus* into Java implementation that uses the JCSP [134, 69] library, a Java implementation of the CSP model for concurrency and communication. However, he reports that the work was ultimately tricky because of the complexity of the abstract syntax and resulting visitor patterns. Thus, the approach was modified in order to explore the translation strategy using Haskell instead. However, the Haskell work did not involve the development of a *Circus* parser.

Yet another approach for model-checking *Circus* was defined by Ye and Woodcock [174], whom defined a link from *Circus* to $CSP \| B$ with model-checking using ProB [135]. However, because the approach splits the *Circus* models into pieces in CSP_M and B, model-checking using other tools such as FDR is not possoble. Moreover, another inconvenience is that such an approach relies solely on ProB, which is a limited tool in terms of processing capabilities: it does not support multiprocessors nor multithreading. Therefore, it is unlikely that ProB would be able to handle larger *Circus* models generated by Ye's tool.

A handmade translation from *Circus* into CSP_M was proposed by Oliveira *et al.* [133], where they define a set of translation rules, which, when combined with the *Circus* refinement laws, result in a subset of *Circus* that can be then translated into CSP_M. Their translating approach requires some attention regarding some *Circus* constructs that are not yet supported by their approach. However, such a translation strategy is limited to the use of a small set of types derived from the same supertype, since polymorphism is not supported by FDR. Moreover the memory model used for capturing the state of a process may lead to state explosion while using a larger set of state variables.

Since CSP_M does not have a notion of variables for a state as in Z, *Circus* or even B-Method, we have to somehow capture them in order to obtain a CSP_M model as similar as possible to the original *Circus* one. Therefore, one could either use a memory model[133, 127] in order to manage the values of the state variables, or else, to adopt the idea of *state-variables parametrised processes*, as used by Beg [16].

A memory model uses an auxiliary process which offers and retrieves the values of each state variables, being executed in parallel with the main execution flow of the system: both processes terminates simultaneously. This approach differs from the one presented by Beg, where the state of a *Circus* process is represented by the process parameters, rather than using a memory cell.

However, the parametrised state proposed by Beg requires the definition of intermediate processes, just as intermediate states in a transition graph. A similar representation using parameters for capturing the state variables in CSP_M was adopted by us previously in the specification of the Integrated Modular Avionics architecture [74], as well as in the formal specification of the haemodialysis machine [75], as a handmade translation strategy. In this case, the state-variables parametrised process approach used in [74] and [75] requires the definition of a large number of CSP_M events and functions for updating an internal *state-process*, which is quite similar to the memory cell used in [133, 127]. However, in our case, we use one *get* and one *set* event for each state variable as well as several auxiliary functions, rather than using an abstraction for mapping variables to values, as presented initially by Mota *et al.*.

2.3 Refinement of Specifications

In our research we also use the concepts of refinement, and therefore, in this section we present an overview of what refinement is, as well as what can be achieved with refinement and what is the range of tools supporting refinement.

The concept of refinement can be explained firstly using the analogy of a drawing process. Initially, the ideas of the final piece are only on the artist's mind, or maybe on a piece of paper from some client's request. At first, he produces a very *abstract* sketch of what he intended to do. Then, he will, step-by-step, use his skills and techniques to introduce *more details* to the drawing, *refining* his *abstract* sketch into a more precise and *concrete* image. Such *refinement* step can occur several times, as the artist aims at the perfection of his piece, and for each pass, the early abstract drawing becomes more concrete until the artist judges it is ready for framing.

By taking the example of the drawing and replacing it with the software development, an abstract specification can be refined into a more concrete implementation, using a *refinement calculus* [9, 8, 121], where details of the intended software are included, sometimes requiring several steps. The refinement process establishes a footpath through the system's development history, where each refinement step may highlight each decision taken on the design and, due to its mathematical nature, the refinement is used for justifying the correctness of each step.

The refinement calculus as presented by Morgan [119] consists of an extension of Dijkstra's guarded command language [52]. In such, a specification statement can be defined as $w : [pre, post]$,

and it can be satisfied by an implementation which, given an initial state that satisfies *pre*, the implementation can be executed resulting in a final state updating the variables in the frame w, that satisfies *post*.

Moreover, the refinement calculus also defines a relation between the refinement steps, known as *refinement relation*. The refinement relation between P and Q, with Q a more concrete model than P, is represented as $P \sqsubseteq Q$ and can be read as P *is refined by* Q. Furthermore, if we look closely to the refinement process, we may see intermediate steps $P \sqsubseteq I_0 \sqsubseteq \ldots \sqsubseteq I_N \sqsubseteq Q$, where $I_0, \ldots, I_N$, may introduce fragments of executable code, as illustrated by Oliveira [129, Chapter 6, p. 129], where *Circus* is refined step-by-step into Java code.

A refinement calculus for Z was introduced by Cavalcanti [34], which was based on Morgan's refinement calculus [120]. Such refinement will be further discussed in Section 3.5, as its implementation and inclusion into our tool, may bring value contribution to the current state-of-art. Moreover, a refinement from Z aiming at obtaining compilable code may be done using a tool like ProZ [135], an extension of ProB. As an advantage, B has notably good tools for model-checking, such as ProB [102], and code generation such as Atelier-B [45] and B-Toolkit [138], whose features are currently lacking for the Z language.

Moreover, Carter *et al.* [33] introduced a strategy for expressing Object-Z in *Perfect* [97], and then, using Perfect Developer [48], refining it into an implementation. Perfect Developer is a software produced by Escher Technologies, which is a tool for fully automatic software verification and code generation to languages like C, C#, and ADA. It automatically generates and discharges all proofs that are necessary to verify the specification in order to guarantee the consistency of the system. The Perfect Developer language introduces the notion of object orientation. It is straightforward to learn for Z users since it has a similar syntax to the Z language, including features such as state invariants and the use of schemas for operations. Similarly, Gomes and Oliveira [77] translated their specification of a cardiac pacemaker [24] written in Z [78], into *Perfect*, and then, refined it into C#, where a graphical user interface (GUI) for the refined code from Perfect Developer was produced. Furthermore, Gomes and Olivera [111] presented a prototype of the pacemaker model using Arduino [6], using a handmade simplified version of the C++ version generated by Perfect Developer.

Among a few works related to the translation of CSP_M into executable code, Zhou and Stiles [178] shows how concurrent programs written in CSP can be translated into sequential programs, using *Mathematica*. Moreover, Raju, Rong and Stiles [137] suggest that CSP_M programs are automatically translated into channel oriented and concurrent executable code, written in Java and C. An experiment for translating CSP||B into Handel-C [30], which is a hybrid language from C and CSP, was presented by Schneider *et al.* [150]. We also know that Welch and Brown present a translation of CSP programs into Java using JCSP [168], while Hilderink presents a translation [80], emphasising in distributed and real-time programs.

There have been several paths for obtaining Java code from CSP models [80]. For instance, Hilderink *et al.* [81] presents their approach for Communicating Java Threads (CJT), while the use of Communicating Sequential Processes for Java (JCSP) [168] is considered in [126, 98, 165, 169]. The differences and similarities between CJT and JCSP are explored in [147]. Besides Java, Welch *et*

al. [167, 166] introduced their approach for obtaining *Occam-π* code from CSP. Other approaches for obtaining *Occam-π* were explored by Moores [118] as well as by Jacobsen and Jadud [91]. Finally, Moores also explored the generation of C code using CCSP [118].

Oliveira [129] developed a refinement calculus for *Circus* currently considered the de-facto reference for *Circus*[1], using tool support with ProofPower-Z [177]. Likewise, Felianchi *et al.* introduced a mechanisation of the Unifying Theories of Programming (UTP) into Isabelle/HOL, supporting *Circus* [58]. Furthermore, Feliachi *et al.* introduced the semantics of *Circus* into Isabelle/HOL, Isabelle/Circus [57]. Finally, another approach to formalise the UTP into Isabelle/HOL was proposed by Foster *et al.* [66].

Another approach for using JCSP as target language from formal specifications is presented by Oliveira *et al.* [134], who present a strategy for the implementation of *Circus* programs in Java. Conversely, from the above-presented approaches, we have here a strategy based in a set of translation rules that are exhaustively applied, transforming programs specified in *Circus* to Java programs using the JCSP library. Freitas and Cavalcanti introduce mechanisation of the *Circus* translation into Java [67]. Moreover, related to code generation from *Circus*, Barrocas and Oliveira introduce *JCircus*, a tool based on *CRefine* [131], which was then extended by Barrocas *et al.* [14], based on the translation rules presented in [129] and also translates *Circus* programs into Java code, with similar ideas to the JCSP implementation.

2.4 Quick *Circus* Guide

The verification of complex and critical systems often requires dealing with a mix of mutable state and communicating processes. Therefore there are needs for formalisms that combine both structural and behavioural aspects of a system, allowing the creation of formal models in a more complete way.

2.4.1 *Circus* Background

A *Circus* specification is in some sense an extension of Z[172] in that it takes the paragraphs of Z and adds new paragraph forms that can define *Circus* channels, processes and actions [2] . Channels correspond to CSP events:

$$\mathbf{channel}\ c : T$$

For both CSP and *Circus*, we usually define a specification comprising a set of paragraphs in the form $N(Expr^{+}) \mathrel{\widehat{=}} A$ where N is a (process/action) name, the $Expr^{+}$ may refer to local parameters, and A is the description of a process/action that may or may not refer to N and the local parameters.

A process is an entity that is willing to perform some events, but not others, depending on its current state. Any processes interact with an environment, also considered as a process. A given process willing to perform an event can be said to be "offering" that event. Whether or not the event occurs depends on both the willingness of the environment to perform it and the synchronisation requirements between the process and its environment.

[1] More details and publications about *Circus* are found at https://www.cs.york.ac.uk/circus/

[2] Which is why our tool was built by extending Jaza[162].

Circus actions can be considered as CSP processes extended with the ability to read and write shared variables, usually defined using a Z schema:

$$LocVars \mathrel{\widehat=} [v_0 : T_x;\ \ldots;\ v_n : T_z \mid inv(v_0, \ldots, v_n)]$$

A *Circus* process is an encapsulation of process-local shared variables and *Circus* actions that access those local variables, along with a 'main' action.

$$
\begin{array}{l}
\mathbf{process}\, P \mathrel{\widehat=} \\
\quad \mathbf{begin} \\
\qquad \mathbf{state}\, State \mathrel{\widehat=} LocVars \\
\qquad P.Actions \mathrel{\widehat=}_{\Delta State} \cdots \\
\qquad \bullet\ \mathbf{var}\, l_0 : U_0;\ \ldots;\ l_m;\ U_m \bullet MA(v_0, \ldots, v_n, l_0, \ldots, l_m) \\
\quad \mathbf{end}
\end{array}
$$

The process P has a state S in which state variables $v_0, \ldots, v_n$ are declared in terms of its respective types $T_x, \ldots, T_z$, and may contain some invariants $inv(v_0, \ldots, v_n)$ regarding the state variables. Moreover, it may have a list of *Circus* actions $P.Actions$ that have access to the state variables and therefore, may update its values. Finally, a main action describes the behaviour of the process. It is possible to have local variables, $l_0, \ldots, l_n$, declared before the main action MA.

Circus processes can only communicate with the external environment via channels. Processes can be modified and combined with each other, using the following CSP operators: sequential composition ($;$), non-deterministic choice ($\sqcap$), external choice ($\Box$), alphabetised parallel ($[\![\ldots]\!]$), interleaving ($|||$), iterated versions of the above (e.g., $\sqcap_{e \in E} \bullet \ldots$), and hiding ($\setminus$).

Circus actions can be built with the CSP operators detailed above, as well as the following CSP constructs: termination (Skip), deadlock (Stop), abort(Chaos), event prefix ($\rightarrow$), guarded action ($\&$), and recursion (μ). Besides, a *Circus* action is defined by a Z schema, or Dijkstra-style guarded commands, including variable assignment ($:=$). Note that actions cannot be defined as standalone entities at the top level of a *Circus* specification.

Also, *Circus* actions can be composed in parallel, but with a difference from standard CSP. Parallel composition of actions requires that we specify for each action, which of the shared variables it is allowed to (visibly) modify. It must be done in such a way that every variable can be modified by at most one arm of the parallel composition. The semantics of a parallel construct is that each side runs on its copy of the shared variables, and the final state is obtained by merging the (disjoint) changes when both sides have terminated. So general parallel composition is given by $P \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, Q$, where ns_1 and ns_2 are the sets of variables that can be modified by P and Q respectively, and cs is the set of channels on which both actions must synchronise.

Finally, *Circus* allows the use of local declarations in a variety of both process and action contexts. For actions, we can declare local variables, using

$$\mathbf{var}\, x : T \bullet A$$

which introduces variable v of type T which is only in scope within A. Variations of these can be used to define parameterised actions, of which the most relevant here is one that supports read-write parameters.

$$(\mathbf{vres}\, x : T \bullet A)(y)$$

Here x is local to A and must differ from y, which is global. At the start, x is initialised with the value of y. Action A then runs, and at the end, y is updated with the final value of x.

2.4.2 *Circus* through an example

We describe the development of a clock alarm, based on the chronometer example from Oliveira's PhD thesis [129], to which we add some alarm features. For model checking purposes, we restrict the type $RANGE$ to values from 0 to 2, instead of the conventional range from 0 up to 59, used for minutes and seconds.

In *Circus*, events are observable and are atomic (either they happen in their entirety, or not at all). Moreover, an atomic event can transfer data through its channel. For instance, an event $c.k$ means that a value k is transported through the channel c. In our example, a few channels are used: the clock *tick* every second; when the *time* is requested, the channel *out* outputs the minutes and seconds as a pair; *radioOn* turns on the radio when the alarm is set; and *snooze* mutes the alarm. The events *snooze* and *radioOn* are part of the added alarm features.

$$
\begin{array}{l}
RANGE == 0 \mathrel{..} 2 \\
ALARM ::= ON \mid OFF \\
\mathbf{channel}\ tick, time, snooze, radioOn \\
\mathbf{channel}\ out : \{min, sec : RANGE \bullet (min, sec)\}
\end{array}
$$

We start the construction of the *Circus* process $WakeUp$ with the definition of the state $WState$ with three state components: *sec* and *min*, for seconds and minutes respectively; and an alarm signal state *buzz*, part of the alarm extension, that can be either ON or OFF.

$$
\begin{array}{l}
\mathbf{process}\ \ WakeUp \mathrel{\widehat=} \mathbf{begin} \\
\quad \mathbf{state}\ WState \mathrel{\widehat=} [sec, min : RANGE;\ buzz : ALARM]
\end{array}
$$

Circus process may contain *Circus* actions, which can describe parts of the process behaviour. Such actions are used in the main action of the process. As an illustration, we present four *Circus* actions, based on the *Chronometer* example. In our example, the first action, $AInit$, sets both *sec* and *min* to zero, as well as *buzz* to OFF. Moreover, $IncSec$ and $IncMin$ increment *sec* and *min* respectively.

$$
\begin{array}{l}
AInit \mathrel{\widehat=} [AState' \mid sec' = 0;\ min' = 0] \\
IncSec \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
IncMin \mathrel{\widehat=} [\Delta AState \mid min' = (min + 1) \bmod 60]
\end{array}
$$

Next, we present a modified version of the *Run* process as an initial attempt in order to model the alarm clock. The action *Run* starts with a *tick*, then it increments *sec* through *IncSec*. Then it behaves like: if $sec = 0$, then it increments *min* through *IncMin*; else if $sec \neq 0$, then nothing happens. We also introduce new features: supposing the alarm is triggered when $min = 1$, then the radio is turned on and the *buzz* component now is ON; if the *time* is requested, the value of *min* and *sec* is delivered through *out* and it skips right after; and finally, the user can press *snooze* and the buzzer is switched

to OFF.

$$Run \mathrel{\widehat=} \left(tick \rightarrow \big(IncSec\big) \; ; \left(\begin{array}{l} \left(\begin{array}{l} \big(sec = 0\big) \;\&\; \big(IncMin\big) \\ \Box \\ \big(sec \neq 0\big) \;\&\; \mathrm{Skip} \end{array} \right) \\ \Box \\ \big(min = 1\big) \;\&\; radioOn \rightarrow (buzz := ON) \\ \Box \\ time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip} \\ \Box \\ snooze \rightarrow (buzz := OFF) \end{array} \right) \right)$$

Finally, the main action of the $WakeUp$ process starts with the initialisation of the state variables, through the action $AInit$, then it recurses the Run action.

$$\bullet\ (AInit \; ; \; (\mu X \bullet (Run \; ; \; X)))$$
end

The Haemodialysis case study [75] uses around 51 channels, includes over 80 actions and comprises about 950 lines of code. Now that we are more familiarised with the *Circus* language, we shall move to the next section where we describe the translation strategy into CSP_M.

2.5 Thesis Proposition

We can see in this chapter that among several formal languages, there are attempts to refine formal specifications into code and also to perform model-checking. However, we can see that there is a lack of automation even though its manual application is very error-prone. It is well known that it is a trend for the industry to apply formal methods in their projects. However, industrial-scale applications would require too much effort and caution from the user in order to avoid the introduction of errors due to the manual work.

In this document, we present the steps towards the development of *Circus2CSP* a tool for model checking *Circus* specifications, using a mechanism that automatically translates *Circus* into CSP_M and from that, uses FDR for model checking. Such a tool is based on the translation strategy proposed by Oliveira *et al.* [133][3], which is limited to a subset of *Circus*. In that work, Oliveira *et al.* use a set of translation rules combined with the *Circus* refinement laws in order to transform a state-rich *Circus* specification into a *stateless* *Circus* version of the specification. For such, the state components of a state-rich *Circus* process are transformed into a *Memory* process [127], with *get* and *set* messages capturing the state changes, resulting in a stateless *Circus* process. We illustrate the summary of our approach in Figure 2.1, compared with the previously mentioned two others from Oliveira *et al.* [133] and Arshad Beg [16].

The entire toolset is developed as an extension of JAZA, which parses Z specifications written in LaTeX, the same input used by the Community Z Tools. *Circus* specifications are written as a sequence of block environments, very similar to the way Z paragraphs are written. LaTeX is a *de facto* standard for writing *Circus* specifications. However, we assume that the *Circus* document is already type checked by existing tools [109]. As part of our contribution, we also improved Oliveira's translation strategy [133], reducing its limitations (restriction to non-polymorphic functions and number of

[3]This work is further detailed in Section 3.1.

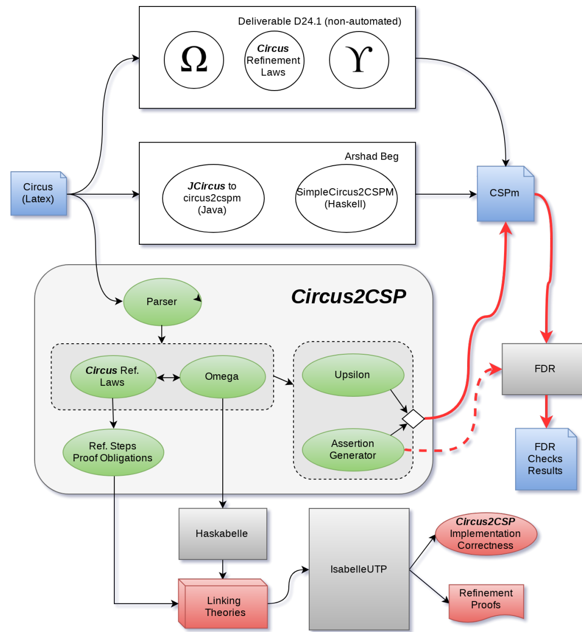


Figure 2.1: Strategies for translating *Circus* into CSP.

state variables supported in the memory model), and including new features such as some translation rules for some Z schemas[4] and the integration of our tool with FDR. Our tool requires that any proof obligation necessary for the refinement, which is not automatically proved, should be proved by the user manually.

Our tool supports most of the *Circus* syntax, but avoids any construct as defined in [132, p. 78]. Furthermore, some features are not yet supported such as: dealing with state invariants nor preconditions in the Z-like schemas; non-determinism of data are not supported; state variables should be initialised prior to the execution of the process; and the consequences of nested parallelism and hidding with non-disjoint name sets were not experimented until now. Finally, the refinement of Z schemas is restricted to those resulting in assignments.

Our goal was to produce a framework using the infrastructure available from Jaza, where the parser for Z was extended and now supports *Circus* and from there, we include new modules like the translation tool and the refinement calculator for *Circus*. Moreover, as shown in Chapter 6, our tool is linked to FDR, and may also be integrated with other tools in the future. Our contribution here is mainly related to the fulfilment of a tool for automatically model-checking *Circus*.

Our focus while model-checking *Circus* is to produce a model in CSP_M where FDR can evaluate using as little computing resources as possible. As such, we provide a refined model from the strategy presented by Oliveira *et al.* [132], where our tool is capable of producing CSP_M models from larger specifications and making it possible for model-checking them using FDR. We highlight that because FDR is a refinement checker, it is not possible to perform temporal logic checks, which is further discussed by Lowe [104].

As a future work, we intend to use a theorem prover in order to verify the correctness of our implementation of our translator. For this reason, we used Haskabelle in order to translate our implementation into the syntax of Isabelle/HOL. We attempted, but did not succeed, to link our syntax

[4]Translation rules specific for those schemas that results in assignments.

with Isabelle/UTP. Our goal was to verify if the models produced using our tool are equivalent to a refinement produced using the *Circus* refinement laws, after having the translation rules from [132] implemented in Isabelle/UTP. However, we encountered restrictions in the expressiveness of Isabelle/UTP for *Circus*, such as the absence of the formalisation of hidding, complex channel communication, and recursive actions, that prevented us from continuing our investigation towards our verified translator. Our preliminary achievements, as well as the issues encountered while attempting to link our tool with Isabelle/UTP, are detailed in Section 8.

Chapter 3

Methodology

In this chapter, we describe our approach for translating *Circus* to CSP_M, where we detail what kind of model we can generate, as well as implementation decisions. We present the ground concepts of *Circus2CSP*, as illustrated in Figure 3.1, a framework that automatically translates *Circus* into CSP_M and later on, model checks these specifications using FDR. Moreover, in Chapter 8, we will discuss why and how we intend to integrate our tool with Isabelle/HOL theorem prover.

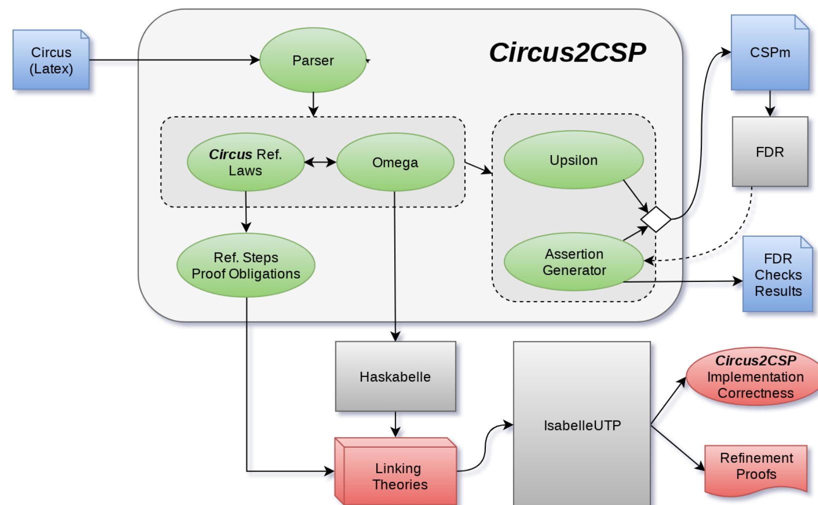


Figure 3.1: Structure of *Circus2CSP*

We first present our work on parsing *Circus* specifications written in LaTeX, by producing an extended version of Jaza [162]. Then we explore a strategy for translating *Circus* into CSP_M. Finally, we discuss the implementation of the translation strategy with a tool written in Haskell.

3.1 Translating *Circus* to CSP_M

Our first attempt model-check the *Circus* haemodialysis specification [75] was to translate it into CSP_M manually and adjust its state-space until the desired checks could be completed. This manual translation was not very satisfactory, as it would have been challenging to make a case that we had done it correctly. Such work motivated us to the development of a mechanised translator that we intend to provide both as a basis for arguing for its correctness, and a high degree of automation to minimise error-prone human interventions.

We started the development of a tool based on the *Circus*-to-CSP_M translation strategy developed for the EU COMPASS project and described in deliverable D24.1 [132, Section 5]. It specifies the translation of a state-rich *Circus* specification into a defined subset of *Circus* by describing functions (Ω_P,Ω_A,Υ) that transform *Circus* syntax. The translation process starts with a state-rich $Circus_{SR}$ process P, which, applying Ω_P (P for *Circus* processes), is then transformed into a state-poor $Circus_{SL}$ process P', in parallel with a *Memory* process, for state and local variable value accesses. We illustrate an overview of the translation in Fig. 3.2.

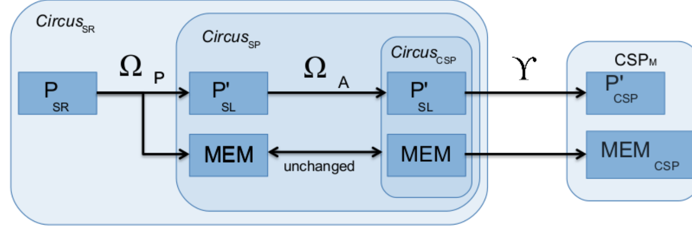


Figure 3.2: Mapping *Circus* into CSP_M (derived from [132, Fig.7, p77])

The Omega functions, Ω_P and Ω_A, move the explicit mutable state from processes and actions respectively, transforming it into a single *Memory* process that is definable in CSP. This process has state parameters that record the current value of every variable and uses *mget* and *mset* channels to provide read and write operations. The memory parameters required are determined by Ω_P when applied to the top-level or 'main' *Circus* process.

In this transformation procedure, the *Circus* refinement laws along with the Ω_A (A stands for *Circus* actions) functions are applied[1] to the main action of the *Circus* process, which is expanded with the definitions of all *Circus* actions of the *Circus* process. Such procedure produces a specification that may be refined using the selected refinement laws. The Ω functions result in models that belong to $Circus_{CSP}$, the CSP subset of *Circus* containing only "state-poor" divergence-free *Circus* processes, ready to be translated into CSP_M by Υ. Finally, the Υ function takes the results of the Ω functions, now corresponding to standard CSP, and renders it in machine-readable CSP_M.

Our tool is able to translate a subset of the *Circus* syntax, and so far, it does not support specifications using Z schemas that would handle complex constructs, other than what can be refined into assignments. Moreover, it does not handle nested parallelism and hidding.

The correctness of the set of Ω translation rules defined by Oliveira *et al.* was proved using the *Circus* refinement laws [132, p. 94]. As part of our research we intended to prove the correctness of the implementation of those translation rules, along with the improvements made througout our research, as will be presented later in this document. In Chapter 8, we present how we we attempted to use Isabelle/UTP, in order to support our approach. We propose to prove that any process P_{SR} in $Circus_{SR}$ is refined by a process P_{SL} in $Circus_{SL}$, using the refinement steps from the Ω functions combined with the *Circus* refinement laws. Therefore, we present the following requirement:

[1] The remainder of the Ω functions are defined in Section 5.3 of the Deliverable report D24.1[132]. Moreover, a list of the *Circus* refinement laws used in the approach can be found in Appendix A of the same document.

Requirement 3.1 *For some set of unprimed and primed state variables, denoting the variables before and after the execution of the process,* P_{SR} *is refined by* $\Omega(P_{SR})$[2].

$$(\exists St, St' \bullet P_{SR}) \sqsubseteq (\exists St, St' \bullet \Omega(P_{SR}))$$

We hightlight that one of the improvements presented in this document is the refinement of the stateless processes where the bindings used for the memory process are no longer defined non-deterministically, and therefore, we are no longer able to prove the equivalence between P_{SR} and $\Omega(P_{SR})$, but but that it is a refinement ($P_{SR} \sqsubseteq \Omega(P_{SR})$). In this case, that is only a refinement, rather than an equality, when not all variables are initialised before any external event. We discuss our approach towards the verified implementation in Chapter 8.

The strategy to be used in the proof of the above presented requirement is based on the fact that the Ω translation is a sequence of refinement steps, using the *Circus* refinement laws. Such an approach is further discussed in Section 8.2. For now, we clarify that we want to decompose the above requirement and attempt to prove that if each step on that sequence of refinement is correct, for example $\Omega_{Step_n} \sqsubseteq \Omega_{Step_{n+1}}$, we also expect to prove that the whole refinement is also correct:

$$P_{SR} \sqsubseteq \ldots \sqsubseteq \Omega_{Step_n} \sqsubseteq \Omega_{Step_{n+1}} \sqsubseteq \ldots \sqsubseteq P_{SL}$$

We note that many of these refinement steps are in fact equivalences, as observed by the *Circus* refinement laws presented in the Appendix D. In the next section we detail the memory model used as part of the translation strategy, as a way of capturing the interactions between the main action of a *Circus* process and its state variables.

3.2 The Memory Model

As it turns out, the architecture of the memory model proved to be crucial to ensuring that the resulting mechanised translation had both wide scopes regarding *Circus* features being supported and resulting in CSP_M models that minimised the time and space workload for the FDR refinement checker.

Initially, our memory model was very similar to that in D24.1, with some differences in naming conventions. In our approach, we defined a notation for renaming the variables allowing the user to identify which are state components, or local variables easily. Variables are renamed by adding a prefix $sv_$ or $lv_$ indicating a state or local variable respectively.

As part of the translation strategy, the CSP_M environment is redefined regarding the type system. Based on the work of Mota *et al.* [127], D24.1 defined a type $UNIVERSE$ comprising any type defined in the specification. Moreover, the names of every state component and local variable are defined as components of $NAME$.

$$[UNIVERSE]$$
$$NAME ::= sv_v_1 \mid sv_v_2 \mid \ldots \mid sv_v_n \mid lv_l_1 \mid \ldots \mid lv_l_n$$

[2]This requirement has not been proven yet, and it will be discussed further in the Future Work section (Section 9.2).

The approach makes use of a set of bindings, $BINDING$, which maps all the names, $NAME$, into the $UNIVERSE$ type.

$$BINDING == NAME \rightarrow UNIVERSE$$
$$\delta == \{sv_v_1 \mapsto T_1, sv_v_2 \mapsto T_2, \ldots, sv_v_n \mapsto T_3, \ldots, lv_l_n \mapsto T_m\}$$

As a result of applying the Ω functions, the state of a *Circus* process is replaced by a $Memory$ action which manages the values, offering $gets$ and $sets$, for the values of the state components of such a process, when translated into CSP_M.

$$\begin{array}{l} Memory \mathrel{\widehat=} \mathbf{vres}\ b : BINDING \bullet \\ \quad (\Box\, n : \mathrm{dom}\, b \bullet mget.n!b(n) \rightarrow Memory(b)) \\ \quad \Box\, (\Box\, n : \mathrm{dom}\, b \bullet mset.n?nv : (nv \in \delta(n)) \rightarrow Memory(b \oplus \{n \mapsto nv\})) \\ \quad \Box\, terminate \rightarrow \mathrm{Skip} \end{array}$$

Such a $Memory$ process runs in parallel with the main action of the translated *Circus* process, and any communication between them occurs through the channels $mget$ and $mset$, where $\delta(n)$ returns the type of the variable n. Moreover, the process execution ends when the $terminate$ signal is triggered. The above three channels compose the MEM_I channel set used hereafter in the translated specification.

$$\mathbf{channelset}\ MEM_I == \{\!|\ mget, mset, terminate\ |\!\}$$

The final specification puts the original process after Ω-translation in parallel with the memory model, synchronising on the MEM_I channels, which are themselves hidden at the top-level, with the binding as a top-level parameter. Note that the semantics of this at the top-level involves a non-deterministic choice of the values in the initial binding b. This results in the following CSP form:

$$\sqcap\, b : BINDING \bullet \left(\begin{array}{l} (\Omega_A(P);\ terminate \rightarrow \mathrm{Skip}) \\ [\![\varnothing \mid MEM_I \mid \varnothing]\!] \\ Memory(b) \end{array} \right) \setminus MEM_I$$

Deliverable D24.1 [132] contains manual proofs of the correctness of the translation [132, Appendix K]. In the next section, we present our approach for translating Z schemas into an equivalent *Circus* construct, in order to provide a translation into CSP_M.

3.3 Extending Jaza

When we started our research, two options were available regarding the existing framework that could be extended in order to implement our translator. Jaza and the CZT. It is well known that CZT is a continuation of work for exploring the Z language and its related/derived formalisms such as TCOZ, Object-Z and *Circus*.

Moreover, one of the mechanisms implemented in CZT is ZLive, an animator for Z, based on the approach of Jaza. However, the CZT was implemented in Java using the visitor pattern. Finally, the CZT development team concluded that the task of implementing the CZT parser for Z was difficult and required over a year [109] to be concluded.

As an illustration of the difficulties for using Java for implementing a tool, Beg [15, p. 71] reports that his attempt for using the CZT was challenging, as he was able to translate only a few simple *Circus* constructs. When he moved to the implementation of complex actions in *Circus*, he reported

that the lack of pattern matching in such actions made him move to implementation of his translator using Haskell.

Either in Beg's work as well as in our Haskell implementation, the pattern matching feature was crucial for implementing our translators. As a consequence, the number of lines of code written was less than what would be needed to implement those functions in Java.

Therefore, we used Jaza [162] as a start point, reusing its existing parser for Z which takes LaTeX specifications as input. Jaza is an open-source tool for animating specifications written in the Z language based on Spivey's Z notation [157]. Its name stands for 'Just Another Z Animator' and was written in Haskell. The current version of the tool, released in 2005, supports parsing and animating programs specified in Z.

Haskell is a purely functional programming language, which means that it prohibits side effects. Moreover, we can have a more clean and elegant code with reduced lines of code for implementing the same rules, compared to an implementation in Java. However, our decision to extend Jaza was not only motivated by the advantages of using Haskell. We also aim at verifying the correctness of the implementation of our translator. For such we can use Haskabelle [79] for translating the Haskell code into the Isabelle-HOL [160] syntax, which is an ML-like language, quite similar to Haskell. Then we can use Isabelle/UTP [66] for the verification.

Details on our journey towards the verification using Isabelle/UTP, as well as the difficulties encountered can be found in Chapter 8. As of now, Isabelle/UTP is not fully able to handle all the *Circus* constructs required for verifying the translation approach, and therefore our implementation. Hence, the verification of our implementation using theorem proving was left as future work.

Currently, there are two notations for the Z language, Spivey [157] Z notation and the ISO/IEC 13568 Standard [89] for the Z language. Our first concern about extending the existing tool for parsing Z so it can support *Circus* is related to the notation used for describing Z specifications. Jaza was written taking into account the Z notation from Spivey. However, the Community Z Tools (CZT) [109] has developed a new framework written in Java based on the ISO Standard. Like Jaza, the CZT framework also reads Z specifications written in LaTeX and parses it w.r.t. the ISO Standard for Z.

In our work, we decided to work using Spivey's Z notation instead of producing a parser compliant with the ISO standard, because ISO Z standard abstract syntax tree has a more complex structure than Spivey's.

3.3.1 Jaza supporting *Circus*

Our first step towards our implementation of the model-checking via CSP_M approach for *Circus* was to extend Jaza's parser. We extended the Haskell code for Jaza and introduced the productions of the *Circus* abstract syntax trees [129]. Since *Circus* is a combination of Z and CSP, we used the existing Haskell code of the Z parser and merged new functions that allow us to parse the *Circus* encoded in LaTeX. We merge both abstract syntax trees since *Circus* includes Z paragraphs, denoted as Par and valid Z identifiers, denoted merely in the *Circus* AST as N.

The Figure 3.3 illustrates an example of a Z specification, a Z state *FactX* which finds the factor of x [157]. Once loaded into Jaza, we can see how one can animate the specifications and assess its execution against a given input. As we can see, Jaza provides some arguments related to underspecification in the case it is unable to provide a correct answer. We assume that the specification

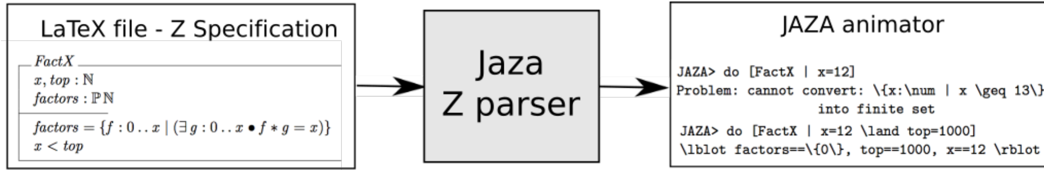


Figure 3.3: Jaza loading LATEX files.

provided is already type checked with existing tools [109]. After parsing the LATEX input, we can identify, on top of the Z paragraphs, the whole structure of *Circus* such as channels, processes, actions, namesets, and channel sets. Moreover, we can evaluate the CSP operations, whose notation differs from each other depending on the scope for *Circus* actions and *Circus* processes.

From this point, we were able to implement the translation approach for *Circus* specifications, as well as the *Circus* refinement calculator, and the mapping functions from *Circus* to CSP_M. We anticipate a few results from our work: during our implementation of the *Circus* parser from the existing Jaza parser for Z was concluded in around four weeks. Moreover, the initial set of the Omega transformations was implemented in two weeks. Finally, the design and implementation of the *Circus* refinement calculator took around three days to be ready to produce some refinement calculations.

We reused the existing definitions of Jaza's Z grammar, and as illustrated below, we introduced our notation for *Circus*. We first introduce how three types of *Circus* paragraphs are defined in the Haskell AST as a subset of the Z paragraphs, *ZPara*, composed by a channel declaration, or a channel set declaration, or a process paragraph in *Circus*,

$$\begin{array}{lcl} ZPara & ::= & \mathbf{channel}\ CDecl \\ & | & \mathbf{channelset}\ N == CSExp \\ & | & \mathbf{process}\ P \mathrel{\widehat=} ProcDef \end{array}$$

can be defined in Haskell with data types as defined below.

```
1  data ZPara =
     | ...
3    | CircChannel [CDecl]
     | CircChanSet ZName CSExp
5    | Process ProcDecl
     | ...
```

The entire syntax of *Circus* is defined in Haskell using datatypes. When it comes to *Circus* actions ($CAction$), we relate the available operators to its representation in the table below.

CircusAST	*HaskellAST*
(S)	`CActionSchemaExpr ZSExpr`
$ZName$	`CActionName ZName`
$Skip$	`CSPSkip`
$Stop$	`CSPStop`
$Chaos$	`CSPChaos`
$Comm \rightarrow C_A$	`CSPCommAction Comm CAction`
$CCommand$	`CActionCommand CCommand`
$(ZPred) \mathbin{\&} C_A$	`CSPGuard ZPred CAction`
$C_A \; ; \; C_A$	`CSPSeq CAction CAction`
$C_A \Box C_A$	`CSPExtChoice CAction CAction`
$C_A \sqcap C_A$	`CSPIntChoice CAction CAction`
$C_A \llbracket ns_1 \mid cs \mid ns_1 \rrbracket C_A$	`CSPNSParal [ZExpr] CSExp [ZExpr] CAction CAction`
$C_A \llbracket cs \rrbracket C_A$	`CSPParal CSExp CAction CAction`
$C_A \vert\vert\vert C_A$	`CSPInterleave CAction CAction`
$C_A \setminus cs$	`CSPHide CAction CSExp`
$C_A(ZExpr^+)$	`CSPParAction ZName [ZExpr]`
$C_A[x/y, z/n]$	`CSPRenAction ZName CReplace`
$\mu N \bullet C_A$	`CSPRecursion ZName CAction`
$(Decl \bullet C_A)(ZName)$	`CSPUnfAction ZName CAction`
$; \; Decl \bullet C_A$	`CSPRepSeq [ZGenFilt] CAction`
$\Box \, Decl \bullet C_A$	`CSPRepExtChoice [ZGenFilt] CAction`
$\sqcap \, Decl \bullet C_A$	`CSPRepIntChoice [ZGenFilt] CAction`
$\llbracket cs \rrbracket \, Decl \bullet \llbracket ns_1 \rrbracket \, C_A$	`CSPRepParalNS CSExp [ZGenFilt] [ZExpr] CAction`
$\llbracket cs \rrbracket \, Decl \bullet C_A$	`CSPRepParal CSExp [ZGenFilt] CAction`
$\vert\vert\vert \; Decl \bullet \Vert[ns_1]\Vert \, C_A$	`CSPRepInterlNS [ZGenFilt] [ZExpr] CAction`
$\vert\vert\vert \; Decl \bullet C_A$	`CSPRepInterl [ZGenFilt] CAction`

In the remainder of this chapter, we turn our attention to the auxiliary mechanisms required for the implementation of the tool. In the next section, we present how we designed the automatic *Circus* refinement calculator.

3.4 Implementing a *Circus* Refinement Laws Calculator

The Ω translation functions makes use of the refinement laws of *Circus* [35], which required us to develop a *Circus* refinement calculator. Our refinement calculator uses a selected subset of the *Circus* refinement laws, where there is a confluence of laws (the laws are applied towards a refined model), and does not introduce any loop during the refinement. The laws are firstly applied to the scope of *Circus* processes, and then, a second iteration applies the laws to the content of the main action of a *Circus* process.

Our implementation of this calculator provides similar functionality to that of CRefine [46]. The critical advantage of reimplementing this functionality in Haskell is that we can, in future work, use Isabelle/HOL in order to verify that our implementation of the laws is correct.

We first introduce the definition of the type constructor *Refinement* for our tool. We define it as a parametrised datatype *Refinement*, where t can either be used for a process refinement ($CProc$) or an action refinement ($CAction$). In Haskell, we define the type `Maybe t` in order to define two possible outcomes of the execution of a given function: it may either contain a certain value y, (herein

represented by `Just y`) or an empty value, (denoted by `Nothing`). We evaluate such a type in order to assert wether or not a refinement step is found during the execution of the calculator. If a refinement step is found, then the tool returns a tagged triple *Done*, that contains the matched instance of that left-hand side (`orig`), along with the corresponding right-hand side (`refined`), plus a list of side-conditions that need to be satisfied (`provisos`). However, when the refinement is not applied to that specification, the calculator returns *None*. Right now, we expect those provisos to be discharged manually by the user. However, our tool will write those provisos in a text file so they can be, in a future, discharged using a theorem prover, like Isabelle/HOL.

```
data Refinement t = Done{orig :: Maybe t, refined :: Maybe t, proviso :: [ZPred] }
                  | None
```

We illustrate our approach with the example of a *Circus* refinement law, **guard combination**, L. 15[3], which allows merging bindings within the context of the action A.

$$(g_1) \mathbin{\&} ((g_2) \mathbin{\&} A) = (g_1 \land g_2) \mathbin{\&} A$$

The above refinement law is implemented in Haskell as the following function *crl_guardComb* that uses pattern matching to check for abstract syntax that matches the left-hand side of the law above, and returns the refinement in *refined*. Moreover, in this refinement law, no proviso is required to be proved. Otherwise, the second clause uses a wild-card pattern for all other cases, which returns `None`.

```
crl_guardComb :: CAction -> Refinement CAction
crl_guardComb e@(CSPGuard g1 (CSPGuard g2 c))
  =  Done{orig = Just e,
          refined = Just (CSPGuard (ZAnd g1 g2) c),
          proviso=[]}
crl_guardComb _ = None
```

The second example we use here to illustrate our implementation of the refinement laws is the **Guard/Parallelism composition distribution** law, L. 19. It extends the guards for the entire parallelism composition in case the initials of the opposite action to the guarded one (action A_2 in our example), is a subset of the channel set cs used for synchronising the two events: $initials(A2) \subseteq cs$.

$$(((g) \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (g) \mathbin{\&} (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$$

The implementation of such law is presented below and behaves as follows: the two first clauses match with the cases where one of either two sides of the parallelism is a guarded command; otherwise, the third clause returns `None`, meaning that no refinement was produced. In either two first clauses, where `Done` is returned, the `proviso` field contains the pending proof obligation regarding the set of initials of the action in parallel with the guarded action.

```
crl_guardParDist :: CAction -> Refinement CAction
crl_guardParDist e@(CSPNSParal ns1 (CChanSet cs) ns2 (CSPGuard g a1) a2)
  =  Done{orig    = Just e,
          refined = Just (CSPGuard g (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2)),
          proviso = [(ZMember (ZTuple [ZSetDisplay (initials a2),
                                      ZSetDisplay (zname_to_zexpr cs)])
                              (ZVar ("\\subseteq",[],[])))]}
crl_guardParDist e@(CSPNSParal ns1 (CChanSet cs) ns2 a1 (CSPGuard g a2))
```

[3]The compilation of the *Circus* refinement laws mentioned in this thesis can be found in the Appendix D.

```
  = Done{orig    = Just e,
        refined = Just (CSPGuard g (CSPNSParal ns1 (CChanSet cs) ns2 a1 a2)),
        proviso = [(ZMember (ZTuple [ZSetDisplay (initials a1),
                                     ZSetDisplay (zname_to_zexpr cs)])
                           (ZVar ("\\subseteq",[],[])))]}
crl_guardParDist _ = None
```

Guided by [132], a relevant subset of the refinement laws have been used in as part of the translation strategy in order to obtain CSP_M specifications from *Circus*. Our refinement calculator was implemented precisely to apply these laws. We intend to ensure that the translation process is as automated as possible in order to produce a CSP_M specification that relies on its original *Circus* version, in order to avoid the introduction of errors in the specification due to user interaction.

An action refinement is performed by the calculator on the main action of a *Circus* process, where the implemented refinement laws are recursively applied through each prefixed action until no further refinement steps are performed. Then, a process refinement is also performed using two other refinement laws, **prom-var-state**, L. 17 as illustrated below, and **prom-var-state2**, L. 18, applied to the context of the *Circus* process itself. The goal of this second refinement step is to promote local variable declarations to the *state* of the *Circus* process. We present two cases of L. 17, defined for processes with an existing **state** paragraph:

$$\begin{array}{l} \textbf{process}\ P \mathrel{\widehat=} \\ \textbf{begin} \\ \quad \textbf{state}\ S \\ \quad L(x : T) \\ \quad \bullet\ (\textbf{var}\ x : T \bullet MA) \\ \textbf{end} \end{array} \quad = \quad \begin{array}{l} \textbf{process}\ P \mathrel{\widehat=} \\ \textbf{begin} \\ \quad \textbf{state}\ S \wedge [x : T] \\ \quad L(_) \\ \quad \bullet\ MA \\ \textbf{end} \end{array}$$

as well as for those without a preexisting **state** paragraph:

$$\begin{array}{l} \textbf{process}\ P \mathrel{\widehat=} \\ \textbf{begin} \\ \quad\quad L(x : T) \\ \quad\quad \bullet\ (\textbf{var}\ x : T \bullet MA) \\ \textbf{end} \end{array} \quad = \quad \begin{array}{l} \textbf{process}\ P \mathrel{\widehat=} \\ \textbf{begin} \\ \quad\quad (\textbf{state}[x : T]) \\ \quad\quad L(_) \\ \quad\quad \bullet\ MA \\ \textbf{end} \end{array}$$

The implementation of the refinement laws is easily achieved because of Haskell's pattern matching feature. For the two cases of the above presented law, we define a function *crl_prom_var_state*, which accepts both parametrised (`ProcDefSpot`) and non-parametrised (`ProcDef`) processes. In both cases, the function calculates the free variables of the main action and renames them with the prefix *lv_*, which are then promoted to the **state** of the process, represented by `(ZSchema (zs++lvs))`, where *zs* is the set of state variables, and *lvs* is the set of local variables promoted from the main action. In any other case where the pattern does not match, `None` is returned meaning that no refinement was performed.

```
crl_prom_var_state :: ZPara -> Refinement ZPara
crl_prom_var_state e@(Process (CProcess p (ProcDef (ProcMain (ZSchemaDef (ZSPlain s []) (ZSchema zs))
    aclst (CActionCommand (CVarDecl x t))))))
  = Done{orig = Just e, refined = Just ref, proviso = []}
      where
        fvs1 = free_var_CAction (CActionCommand (CVarDecl x t))
        fvs2 = free_var_CAction t
        ffvs = diff_varset fvs2 fvs1
        -- prefix the local variables with lv_
        lvs = rename_genfilt_lv p x
        nl = rename_list_lv p (varset_to_zvars ffvs) x
```

```
        subs = make_subinfo nl fvs2
12      -- rename the occurrences of local variables
        -- as prefixed in the main action
14      finalsubs = sub_CAction subs ma2
        ref = (Process (CProcess p (ProcDef
16                (ProcMain (ZSchemaDef (ZSPlain s []) (ZSchema (zs++lvs))) aclst finalsubs))))
crl_prom_var_state e@(Process (CProcess p (ProcDefSpot yR (ProcDef
18                          (ProcMain (ZSchemaDef (ZSPlain s []) (ZSchema zs))
                                        aclst
20                                      (CActionCommand (CVarDecl x t)))))))
  = Done{orig = Just e, refined = Just ref, proviso = []}
22    where ...
crl_prom_var_state _ = None
```

We now explain the behaviour of the internal mechanism of our refinement calculator. As mentioned before, a *Circus* specification in *Circus2CSP* consists of a list of Z paragraphs, and, for each *Circus* process within that list, the refinement calculator will attempt to apply each of the refinement laws implemented. In case the result from applying a law is different from the original model, it means that a refinement step has occurred and the tool recurses trying to apply the refinement laws to the new refined model. The refinement is completed when no new refinement step is found, represented by `None`.

Similarly, the tool applies all the refinement laws implemented in the context of *Circus* actions. The difference here is that the mechanism is more complicated since the tool first attempt to apply the refinement to the top level of the action, and instead of stopping when no further refinement is found to that construct, it recurses over the branches of the operators.

For example, let's use the action $A_1 \Box A_2$. The first step is to attempt to refine that action, and, whenever the tool returns `None`, it will then recurse, attempting to refine both A_1 and A_2. Finally, it stops when no further refinement is found when applying the laws to the branches of the action. Our implementation in Haskell has demonstrated that even using a larger specification, the refinement laws are applied quickly to both processes and actions levels, in just a few milliseconds.

3.4.1 A simple refinement example

We introduce here an example of the refinement steps of a *Circus* action, where the refinement steps are illustrated below. In the following example, assuming that $A1 \mathrel{\widehat=} c1 \rightarrow Skip$ and $A2 \mathrel{\widehat=} c2 \rightarrow Skip$, we want the calculator to apply L. 19 and L. 15, as presented above, in order to bring all the guards together as a conjunction, $v1 = 0 \land v2 = 0$.

$$
\begin{array}{ll}
(v1 > 0) \mathbin{\&} (\ (v2 > 0) \mathbin{\&} A1 \llbracket \varnothing \mid \{\!| c1, c2 |\!\} \mid \varnothing \rrbracket A2\ ) & \\
= & [\text{L. } 19 - \text{prov:} \mathit{initials}(A2) \subseteq \{\!| c1, c2 |\!\}] \\
(v1 > 0) \mathbin{\&} (\ (v2 > 0) \mathbin{\&} (\ A1 \llbracket \varnothing \mid \{\!| c1, c2 |\!\} \mid \varnothing \rrbracket A2\ )\ ) & \\
= & [\text{L. } 15] \\
(v2 > 0 \land v1 > 0) \mathbin{\&} (\ A1 \llbracket \varnothing \mid \{\!| c1, c2 |\!\} \mid \varnothing \rrbracket A2\ ) &
\end{array}
$$

The calculator generates the refinement steps of the above action in the Haskell AST notation for *Circus*. The resulting calculation can be exported into a text file where the user can check each interaction. We illustrate below the extract of the output file from the above *Circus* refinement steps.

```
1  CSPGuard
```

```
  (ZMember (ZTuple [ZVar ("v1",[],"NAT"),ZInt 0]) (ZVar (">",[],"")))
  (CSPNSParal NSExpEmpty (CChanSet ["c1","c2"]) NSExpEmpty
    (CSPGuard (ZMember (ZTuple [ZVar ("v2",[],"NAT"),ZInt 0]) (ZVar (">",[],"")))
              (CActionName "a1"))
    (CActionName "a2"))
=   provided[ZMember (ZTuple [ZSetDisplay [ZVar ("c2",[],"")],
                          ZSetDisplay [ZVar ("c1",[],""),ZVar ("c2",[],"")]])
                    (ZVar ("\\subseteq",[]))]
CSPGuard
  (ZMember (ZTuple [ZVar ("v1",[],"NAT"),ZInt 0]) (ZVar (">",[],"")))
  (CSPGuard (ZMember (ZTuple [ZVar ("v2",[],"NAT"),ZInt 0]) (ZVar (">",[],"")))
      (CSPNSParal NSExpEmpty (CChanSet ["c1","c2"]) NSExpEmpty (CActionName "a1") (CActionName "a2")))
=   provided[none]
CSPGuard
  (ZAnd (ZMember (ZTuple [ZVar ("v1",[],"NAT"),ZInt 0]) (ZVar (">",[],"")))
        (ZMember (ZTuple [ZVar ("v2",[],"NAT"),ZInt 0]) (ZVar (">",[],""))))
  (CSPNSParal NSExpEmpty (CChanSet ["c1","c2"]) NSExpEmpty (CActionName "a1") (CActionName "a2"))
```

In the next section, we present one of our contributions to the existing work: we list a few rules for translating a subset of Z schemas, which was not included in the translation strategy presented by Oliveira *et al.* [50].

3.5 Rewriting Z Schemas into *Circus* Actions

In this section, we devote ourselves to look for an approach for refining those Z schemas that can be refined into assignements in *Circus*. We know that producing a general refinement set of rules is a challenging task and in this work, we give a few ideas of how to capture essentially operations that modify the state variables, which are then refined into assignments.

We propose in this section the refinement of only a few cases which can be commonly found in the literature. The scope of our research in extending the translation rules gives us the ability to explore the expressiveness of using Z schemas for dealing with the process state variables.

However, while exploring the world of Z schemas, we are aware that we are not fully able to give support to all the properties checked during the refinement checks when model-checking using FDR. For instance, the translation strategy introduced by Oliveira *et al.* does not take the state invariants into account. Furthermore, the semantic model of *Circus* as presented in [129] does not explicitly mention invariants. Moreover, capturing the state invariant as well as checking for any violation in CSP_M might make the job of the model checker extremely difficult since the calculation while exploring the state space might be much harder and would require more computing resources. Finally, when translating into CSP_M, the operations over the state variables are transparent (hidden) to the environment, and therefore, nothing can be said about invariants, since our focus is on model checking the system behaviour and not the data aspects of the system.

Our proposed rules for refining some of the constructs available in Z when using schemas are based on the Z Refinement Calculus [34], which in its turn is based on Morgan's refinement calculus [119, 121].

We produce a transformation of Z schemas considering that they were declared in the normal form, $St \mathrel{\widehat=} [d \mid inv]$. Then, we use the transformation from schemas to specification statements, using the conversion laws [34, p. 55]. A specification statement is defined regarding its precondition, which must be satisfied by the initial state, and if so, then the postcondition is satisfied by the resulting state from changing the variables in the frame. Such statement can be defined as $fl : [pre, post]$, where fl are the variables from the frame list whose values are changed in that operation. Therefore, the basic

conversion (bC) from Cavalcanti transforms a schema that modifies the state, St, into a specification statement:

$$[\Delta\, St;\ di?;\ do! \mid p] \ =_{bC}\ \alpha\, d, \alpha\, do! : [inv \wedge \exists\, d';\ do! \bullet inv' \wedge p, inv' \wedge p]$$

However, the notation from Oliveira [129] suggests that a schema can be translated into a specification statement after a normalisation, where the invariant is included in the predicate p of the schema.

$$\begin{aligned}
&[\Delta\, St;\ di?;\ do! \mid p] \\
&= && [\text{normalisation}] \\
&[St;\ St';\ di?;\ do! \mid inv \wedge p] \\
&= && [[129, \text{def B.40}]] \\
&\alpha\, St, \alpha\, do! : [\exists\, St';\ do! \bullet inv' \wedge p, inv' \wedge p]
\end{aligned}$$

Then, from a specification statement, an omega translation rule from [132, p. 91] might be used in order to produce a *Circus* action compatible to the syntax that can be translated into CSP_M.

$$\begin{array}{l}
\Omega_A \left(w : \left[\begin{array}{l} pre(v_0, \ldots, v_n), \\ post \left(\begin{array}{l} v_0, \ldots, v_n, l_0, \ldots, l_n \\ v_0', \ldots, v_n', l_0', \ldots, l_n' \end{array} \right) \end{array} \right] \right) \mathrel{\widehat=} \\
\quad mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\
\quad mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_n?vl_n \rightarrow \\
\quad \left(\begin{array}{l}
\neg\, \big(pre(vv_0, \ldots, vv_n, lv_0, \ldots, lv_n)\big) \ \&\ Chaos \\
\Box\big(pre(vv_0, \ldots, vv_n, lv_0, \ldots, lv_n)\big) \& \\
\quad \left(\begin{array}{l}
\sqcap\, vv : \left\{ \begin{array}{l}
x_0 : \Gamma(v_0);\ \ldots;\ x_n : \Gamma(v_n); \\
\quad x_{n+1} : \Gamma(l_0);\ \ldots x_m : \Gamma(l_m) \\
\mid post \left(\begin{array}{l} vv_0, \ldots, vv_n, vl_0, \ldots, vl_n \\ x_0, \ldots, x_n, x_{n+1}, \ldots, x_m \end{array} \right) \\
\wedge\, \overline{w}' = \overline{w} \\
\bullet\ (x_0, \ldots, x_n, x_{n+1}, \ldots, x_m)
\end{array} \right\} \bullet \\
\quad mset.v_0!(vv.0) \rightarrow \ldots \rightarrow \\
\quad mset.v_0!(vv.n) \rightarrow \\
\quad mset.l_0!(vv.(n+1)) \rightarrow \ldots \rightarrow \\
\quad mset.l_m!(vv.(n+1)) \rightarrow \\
\quad Skip
\end{array} \right)
\end{array} \right)
\end{array}$$

As mentioned earlier in this section, producing a model that preserves the invariants during its operations makes the task of model checking more expensive. The translation rule for specification statement from [132, p. 91] introduces a non-deterministic choice over all the possible combination of values for the frame variables that would preserve the invariants along with a statement that any other variable not mentioned in the frame is unchanged. However, the validation of the translation rule for the specification statement is not complete in [132].

We propose here an initial attempt to capture the behaviour of Z schema in a more concrete model, compared to the generalised translation rule presented above, from [132, p. 91]. In such work, where schemas can be translated into specification statements and then, the later can be translated into an external choice between either chaos or an update to the memory process depending on whether or not the preconditions are satisfied. One of the constructs we are most interested in are the state values updates using schemas, which can be refined into assignments using Z Refinement Calculus [34]. This would eliminate the task of translating the predicates into a CSP_M notation that can be interpreted

and evaluated in FDR. Cavalcanti [34, p. 140] defines a refinement from Z schemas that modifies a state into assignments, provided that the invariant is not violated when the values from the variables are updated:

$$[\Delta\, St;\ di?;\ do! \mid c_1' = e_1 \land \ldots \land c_n' = e_n \land o_1! = e_{n+1} \land o_m! = e_{n+m}]$$
$$\sqsubseteq \qquad [assC \text{ - } \mathbf{provided}\ inv[e_1, \ldots, e_n / c_1, \ldots, c_n] \text{ and } c_1', \ldots, c_n' \notin FreeVars(e_1, \ldots, e_{n+m})]$$
$$c_1, \ldots, c_n, o_1!, \ldots, o_m! := e_1, \ldots, e_{n+m}$$

If we restrict the above definition to schemas that has no input or output variables, by applying $assC$, we obtain an assignment.

$$[\Delta\, St \mid c_1' = e_1 \land \ldots \land c_n' = e_n]$$
$$\sqsubseteq \qquad [assC \text{ - } \mathbf{provided}\ inv[e_1, \ldots, e_n / c_1, \ldots, c_n] \text{ and } c_1', \ldots, c_n' \notin FreeVars(e_1, \ldots, e_n)]$$
$$c_1, \ldots, c_n := e_1, \ldots, e_n$$

To this date, we assume here that the provisos are proved to be correct through proofs by the user. However, in the future work, such task might be supported by our tool with a link to Isabelle/HOL theorem prover, for discharging the proof obligations, such as those we intend to automate with our refinement calculator. We address our effort for achieving this in Section 8.

One of the requirements when model-checking a system is to produce a model whose range of values is enough for covering any condition imposed by an operation. However, when including the state invariant, we are also restricting the range of values permitted to be used within the system. From the example of the chronometer in Section 2.4, we know that both *min* and *sec* was declared as natural numbers. However, while thinking of a chronometer in the real world, we know that neither a second, nor a minute goes beyond 59 units, without flipping the next unit counter. Therefore, while model-checking the chronometer model in CSP_M, it is safe to restrict the range of *min* and *sec* to $0 \mathinner{..} 60$, where 60 is an unexpected value in the system.

However, if one makes the *Circus* model more restricted, and declare the state as $[min, sec : \mathbb{N} \mid min < 60 \land sec < 60]$, we know that the semantics of *Circus* will preserve the invariant over its behaviour. However, depending on the invariants used, the translation into a CSP_M model might become difficult since we do not have an easy way of modelling complexes predicates from Z in CSP_M.

We experimented with the impact of explicitly including invariant and precondition checks in *Circus* models and translating them into CSP_M using the example of the Chronometer, from Section 2.4, with a new process *Chrono*. When using the translation rules presented in [132], we identified that it is hard for FDR to check the model. The chronometer example presented was translated using the conversion from normalised schemas to specification statement and from there, to the appropriate rules that introduce a condition to whether or not the *pre* is satisfied or not. In case of being satisfied, it behaves as a non-deterministic choice of all possible combinations of values from the state variables which satisfies both the invariant and the precondition, followed by updating these values in the memory model. Otherwise, should *pre* is not satisfied, it behaves such as *Chaos*.

Our example of the chronometer has only two state variables and the results obtained using FDR are enough to show how the invariant checks throughout the specification increase the time spent

during the assertion check in FDR. We deliberately modified the original model with the inclusion of the state invariant restricting both *min* and *sec* to values below 60, in order to experiment the translated model in FDR.

$$
\begin{array}{l}
\textbf{process } Chrono \mathrel{\widehat=} \\
\quad \textbf{begin} \\
\quad \textbf{state } AState \mathrel{\widehat=} [sec, min : \mathbb{N} \mid min < 60 \land sec < 60] \\
\quad AInit \mathrel{\widehat=} [AState' \mid sec' = 0;\ min' = 0] \\
\quad IncSec \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
\quad IncMin \mathrel{\widehat=} [\Delta AState \mid min' = (min + 1) \bmod 60] \\
\quad Run \mathrel{\widehat=} \left(\begin{array}{l} tick \rightarrow \big(IncSec\big); \\ \left(\begin{array}{l} \big(sec = 0\big) \mathbin{\&} \big(IncMin\big) \\ \Box \\ \big(sec \neq 0\big) \mathbin{\&} \mathrm{Skip} \end{array} \right) \\ \Box\ time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip} \end{array} \right) \\
\quad \bullet\ AInit\ ;\ (\mu X \bullet Run\ ;\ X) \\
\quad \textbf{end}
\end{array}
$$

We illustrate our experiment in Table 3.1 while exploring the inclusion of state invariants and precondition verification in the chronometer model, and used the following derived models[4]:

*D*241 Model translated using the approach from [132] without the inclusion of invariants and preconditions, using a non-deterministic choice of any possible set of bindings.

*D*241*Inv* Model translated using the approach from [132] including the invariants as a restriction to the bindings set.

*D*241*Pre* Model translated using the approach from [132] which includes precondition checks before the operations, but does not include the invariants. The preconditions, in this case, includes the invariant, as defined for the translation rule presented by Oliveira [129], where the schemas are transformed into $\alpha\, St, \alpha\, do! : [\exists\, St';\ do! \bullet inv' \land p, inv' \land p]$, where the invariant is part of the precondition.

*D*241*InvPre* Combination of *D*241*Inv* and *D*241*Pre*.

CTOC Model translated using our improved translation rules, the result from our tool *Circus2CSP*, as will be discussed in Section 4.1 (no invariant checks).

CTOCPre Extension of *CTOC* model but with the same translation rules for *D*241*Pre*, including the precondition checks.

From the above listed models, our tool is able to automatically generate *CTOC*, whereas the others were generated by hand. We perfomed checks for deadlock freedom[5] using the translated models using the six variants presented above combined with a different range of values for the natural number, replacing the restriction to the value 60, for model checking purposes. For example, in a specification where the values for natural numbers are restricted to the range $0 \mathinner{..} 10$, the process state is then defined as $[min, sec : 0 \mathinner{..} 10 \mid min < 10 \land sec < 10]$.

[4]The files used in this experiment can be found in the tool repository, and from there, are located in the path "https://bitbucket.org/circusmodelcheck/circus2csp/exs/cases/alarm/gCSPm/tests/"

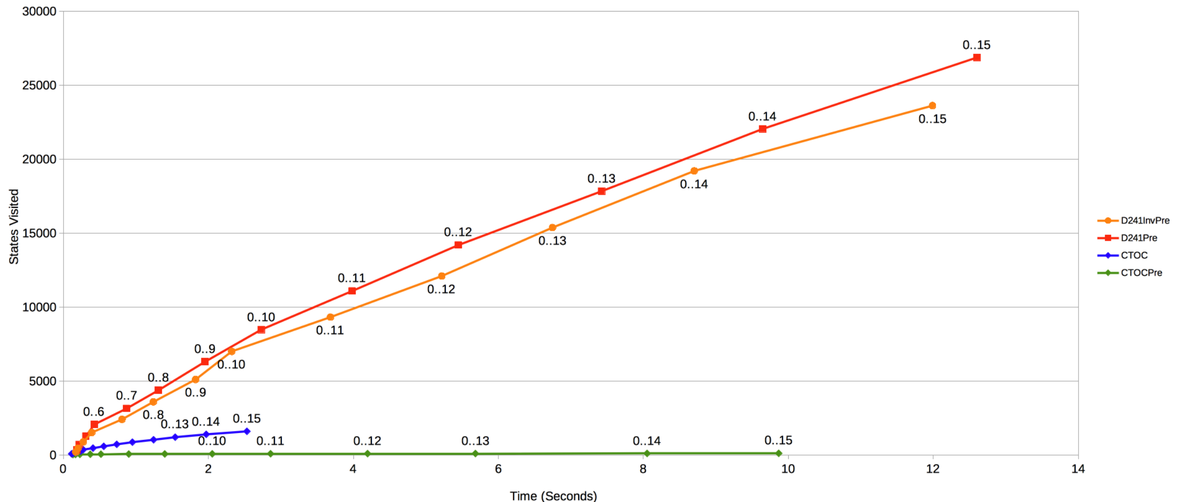
[5]The tests were performed using *Intel Core i7* 2.8GHz CPU with 16GB of RAM.

Table 3.1: Interference of invariants and preconditions in CSP_M- Deadlock freedom checks

	CTOC		CTOCPre		D241		D241Inv		D241InvPre		D241Pre	
Values Range	Exec Time	States Visited	Exec Time	States Visited	Exec Time	States Visited	Exec Time	States Visited	Exec Time	States Visited	Exec Time	States Visited
0..3	0.116	68	0.134	21	0.206	1085	0.177	610	0.173	190	0.187	337
0..4	0.138	118	0.169	28	0.212	2946	0.183	1885	0.204	449	0.219	701
0..5	0.186	182	0.229	35	0.346	6547	0.255	4546	0.276	876	0.312	1261
0..6	0.242	260	0.373	42	0.416	12734	0.35	9355	0.393	1.513	0.428	2059
0..7	0.275	352	0.52	49	0.631	22521	0.545	17242	0.81	2.402	0.874	3137
0..8	0.411	458	0.904	56	0.878	37090	0.705	29305	1.242	3585	1.312	4.37
0..9	0.559	578	1.4	63	1.158	57791	1.138	46810	1.826	5104	1.955	6301
0..10	0.738	712	2.055	70	1.687	86142	1.482	71191	2.323	7001	2.73	8471
0..11	0.954	860	2.859	77	2.147	123829	1.954	104050	3.685	9318	3.986	11089
0..12	1.246	1022	4.197	84	2.714	172706	2.57	147157	5.22	12097	5.45	14197
0..13	1.544	1198	5.684	91	3.68	234795	3.308	202450	6.748	15380	7.424	17837
0..14	1.972	1388	8.052	98	4.955	312286	4.356	272035	8.702	19209	9.646	22051
0..15	2.533	1592	9.867	105	5.846	407537	5.452	358186	11.988	23626	12.6	26881
0..60	3m27s	25262	22m29s	1024	2h48m	99M	01h40m	91M	52m28s	3.7M	65m22s	3.8M

The first difference in the results is that from the models $D241$ and $D241Inv$, and therefore, those that do not restrict the behaviour of the system to whether or not the preconditions are satisfied, and therefore neither the precondition itself nor the state invariants are being checked. For those cases, the number of states visited was over 10-fold more significant than the other cases, as illustrated in Fig. 3.4. We observe that the results from $D241$ and $D241Inv$ were so significant, that they do not allow us to see the results obtained for $CTOC$ and $CTOCPre$. The labels in the chart refer to the value ranges of natural numbers, the first column from Table 3.1.

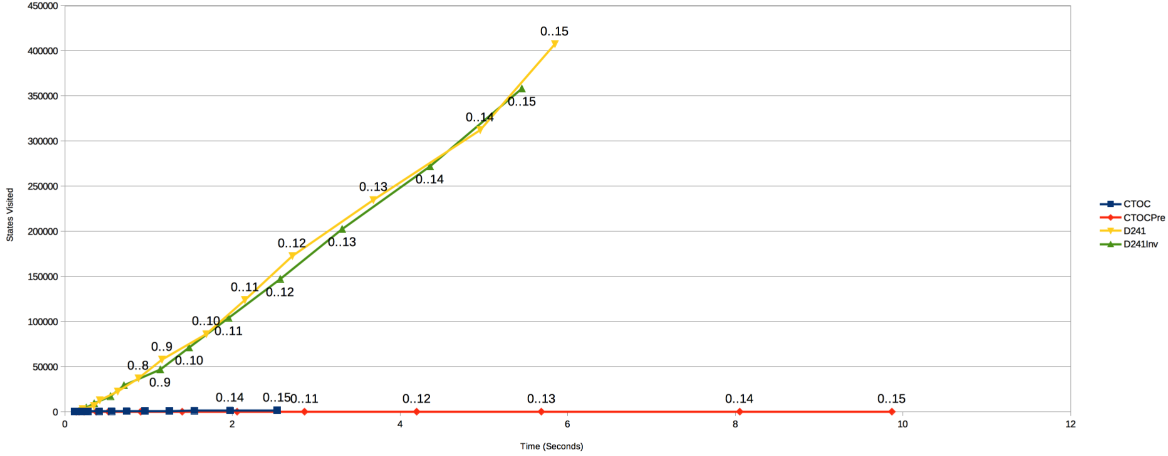
Figure 3.4: Experiment results derived from Table 3.1 - $D241$, $D241Inv$, $CTOC$ and $CTOCPre$.



However, the influence of a precondition check within an operation makes a significant reduction in the state exploration. Moreover, by restricting the bindings to the invariants of the state, the results were quite similar to the ones without such restriction. Finally, the results from the models $D241InvPre$ and $D241Pre$ shows that the state space explored was smaller, but the time consumed by FDR for calculating the assertions has doubled, as illustrated in the Fig 3.5, generated from the data collected in Table 3.1. Note that for clarity, that chart does not include the results of $D241$ and $D241Inv$.

We also noticed that all the above presented result were executed in a much large time frame than the approaches using the translation from our tool, *Circus2CSP*. However, the models generated by our tool does not include neither invariants nor preconditions. Moreover, even when including the precondition checks ($CTOCPre$) within the model generated, with the manual inclusion of preconditions, the number of visited states reduced considerably, but with the cost of 4-fold longer execution

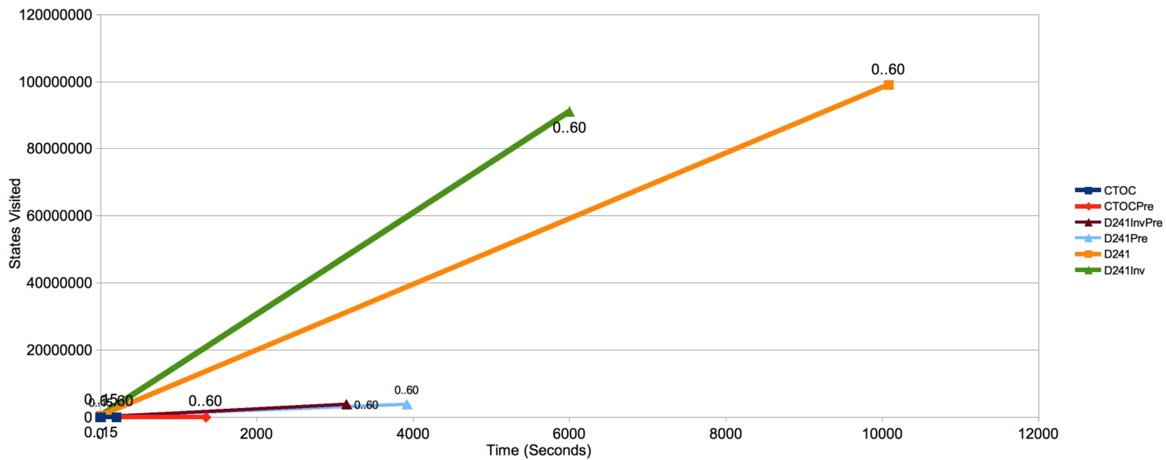
Figure 3.5: Experiment results derived from Table 3.1 - $D241InvPre$, $D241Pre$, $CTOC$ and $CTOCPre$.



than the case of $CTOC$.

As a way of experimenting the real world example of the chronometer, we experimented the model ranging from 0 up to 60, whose values are presented in the last row of Table 3.1, illustrated in the figure below. In general, the CSP_M models ($CTOC$) translated using our tool were evaluated by FDR using a much smaller state space and were checked in less time than all the other models experimented.

Figure 3.6: Experimenting real chronometer values (0 up to 60) derived from Table 3.1



We see a significant difference among the results from the approaches evaluated, where the model using $CTOC$ was evaluated (3 minutes) by FDR in 98% less time than the time spent to check the model using $D241$ (over 2h48). Such a result shows how different models of the same system can be affectedby the checks of invariants and preconditions, as well as how optimising the memory model can result in much smaller state exploration when using FDR. Finally, we observed no correlation between time and state visited, inspite the use (or not) of compression by default in FDR.

3.5.1 An Example of the Transformation

As an example of the transformation of Z schemas to fit into our translation, we use the *Chrono* process as defined previously in this section, which is based on the example from Oliveira's PhD

thesis [129, Section 1.3, p.6]. It simply updates the minutes and seconds, as a normal clock, after each *tick*. Moreover, it also may display, through the signal *out*, the current time, should a signal *time* occur.

$$
\begin{array}{l}
RANGE == \{0 \mathinner{\ldotp\ldotp} 59\} \\
\mathbf{channel}\ tick, time \\
\mathbf{channel}\ out : RANGE \times RANGE \\
\mathbf{process}\ Chrono \mathrel{\widehat=} \\
\quad \mathbf{begin} \\
\quad \mathbf{state}\ AState \mathrel{\widehat=} [sec, min : \mathbb{N} \mid min < 60 \land sec < 60] \\
\quad AInit \mathrel{\widehat=} [AState' \mid sec' = 0;\ min' = 0] \\
\quad IncSec \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
\quad IncMin \mathrel{\widehat=} [\Delta AState \mid min' = (min + 1) \bmod 60] \\
\quad Run \mathrel{\widehat=} \left(\begin{array}{l} tick \rightarrow \big(IncSec\big); \\ \left(\begin{array}{l} \big(sec = 0\big) \mathbin{\&} \big(IncMin\big) \\ \Box \\ \big(sec \neq 0\big) \mathbin{\&} \mathrm{Skip} \end{array}\right) \\ \Box\ time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip} \end{array}\right) \\
\quad \bullet\ AInit\ ;\ (\mu X \bullet Run\ ;\ X) \\
\mathbf{end}
\end{array}
$$

As part of the translation strategy, all *Circus* actions are promoted to the main action. Therefore, after applying the rules presented in the previous section, we obtain the following *Circus* specification, which will then be translated, using the omega (Ω) functions, into a *Circus* syntax which can be translated into CSP_M.

$$
\begin{array}{l}
\mathbf{process}\ \ Chrono \mathrel{\widehat=} \\
\mathbf{begin} \\
\quad \mathbf{state}\ AState \mathrel{\widehat=} [sec, min : \mathbb{N} \mid min < 60 \land sec < 60] \\
\quad \bullet \left(\begin{array}{l} (sec, min := 0, 0); \\ \left(\begin{array}{l} \mu X \bullet \\ \left(\begin{array}{l} tick \rightarrow (sec := (sec + 1) \bmod 60); \\ \left(\begin{array}{l} \big(sec = 0\big) \mathbin{\&} (min := (min + 1) \bmod 60) \\ \Box \\ \big(sec \neq 0\big) \mathbin{\&} \mathrm{Skip} \end{array}\right) \\ \Box\ time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip} \end{array}\right);\ X \end{array}\right) \end{array}\right) \\
\mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{process}\ \ AChrono \mathrel{\widehat=} b_RAN : RANGE\ \bullet \\
\mathbf{begin} \\
\quad Memory \mathrel{\widehat=} \ldots \\
\quad \bullet \left(\begin{array}{l} mset.sv_sec.(RAN.0) \rightarrow mset.sv_min.(RAN.0) \rightarrow \\ \left(\begin{array}{l} \mu X \bullet \\ \left(\begin{array}{l} tick \rightarrow mset.sv_sec.(RAN.(v_sv_sec + 1) \bmod 3) \rightarrow \\ mget.sv_min?v_sv_min : (\delta_{RAN}(sv_min)) \rightarrow \\ mget.sv_sec?v_sv_sec : (\delta_{RAN}(sv_sec)) \rightarrow \\ \left(\begin{array}{l} \big(v_sv_sec = 0\big) \mathbin{\&} \\ \quad mset.sv_min.(RAN.(v_sv_min + 1) \bmod 3) \rightarrow \mathrm{Skip} \\ \Box \\ \big(v_sv_sec \neq 0\big) \mathbin{\&} \mathrm{Skip} \end{array}\right) \\ \Box\ time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip} \end{array}\right);\ X \end{array}\right) \end{array}\right) \\
\mathbf{end}
\end{array}
$$

The proofs supporting the above translation can be found in Appendix G.1.

3.5.2 Final considerations

We see here a future link of the Z refinement calculus [34] with our tool. Currently, our tool supports only the refinement of those schemas that can be translated into assignments. Moreover, the refinement laws for Z [34] are different from the refinement laws for *Circus* [129]. We have some ideas for a future implementation of the refinement of other constructs, which might also be advantageous. For instance, schema conjunction may be refined into a sequential composition of schemas. Moreover, schema disjunction might be refined into alternation where the precondition of schemas is transformed into guards.

Our experiments presented in this section, as well with the other presented in this document, show the potential for the tool and further investigation on the effectiveness of existing refinement and translation approaches when it comes to the implementation of tools for supporting them. Moreover, the scalability of the approaches should be considered through empirical research, which might be achieved with experiments such as the ones we produced throughout our research, as we report in this document. We discuss more on how we plan to extend our contributions to this work in Section 9.2, where we discuss what can be used and how we intend to integrate our refinement calculator and Z schemas refinement between our tool and theorem provers.

In the next section, we introduce the whole translation structure from state-rich *Circus* processes to state-poor *Circus*, using the Ω functions. Moreover, we also present our findings regarding the translation strategy initially presented in [132], and provide several improvements towards the automatic translator tool.

Chapter 4

Rewritting *Circus* state-rich processes with Ω

The translation steps using the Ω functions involve the use of *Circus* refinement laws, as illustrated in Fig. 3.2. We now present each step of the transformations towards the state-poor *Circus* process that can then be translated into CSP_M. The steps here are present in the Deliverable 24.1 [132], and the steps presented beyond Step 4.6 are part of our contribution to state of the art, as our research has demonstrated that these steps would result in a more efficient specification regarding model-checking. In order to demonstrate how the translation scheme affects a *Circus* process, we will again use the example of the process P, from the Section 2.4.2, as a starting point for the translation process.

$$\begin{array}{l}\mathbf{process}\ P \mathrel{\widehat=} \\ \mathbf{begin} \\ \quad \mathbf{state}\ State \mathrel{\widehat=} [v_0 : T_0;\ \ldots;\ v_n : T_n \mid inv(v_0, \ldots, v_n)] \\ \quad P.Actions \mathrel{\widehat=}_{\Delta} P.State \\ \quad \bullet\ MA(v_0, \ldots, v_n, l_0, \ldots, l_m) \\ \mathbf{end}\end{array} \tag{4.1}$$

Given the initial shape of the process P, the first iteration is to expand the definitions of all the $P.Actions$ into MA, and therefore, the process will no longer have any other action defined but the main action. Then, an action refinement in the main action of the process, MA, is executed, where the *Circus* refinement laws are used in order to promote any local variable towards the outermost side of the main action, using laws such as Law 16[1]. Moreover, all local variables are then renamed in order to avoid name clashes in the next transformation steps.

$$\begin{array}{l}\mathbf{process}\ P \mathrel{\widehat=} \\ \mathbf{begin} \\ \quad \mathbf{state}\ State \mathrel{\widehat=} [v_0 : T_0;\ \ldots;\ v_n : T_n \mid inv(v_0, \ldots, v_n)] \\ \quad \bullet\ \mathbf{var}\ l_0 : U_0;\ \ldots;\ l_m;\ U_m \bullet MA(v_0, \ldots, v_n, l_0, \ldots, l_m) \\ \mathbf{end}\end{array} \tag{4.2}$$

The next step is the execution of a *Circus* process refinement, where any local variable is then promoted into state components, using the Law 17 and Law 18, respectively.

$$\begin{array}{l}\mathbf{process}\ P \mathrel{\widehat=} \\ \mathbf{begin} \\ \quad \mathbf{state}\ State \mathrel{\widehat=} [v_0 : T_0;\ \ldots;\ v_n : T_n;\ l_0 : U_0;\ \ldots;\ l_m;\ U_m \mid inv(v_0, \ldots, v_n)] \\ \quad \bullet\ A(v_0, \ldots, v_n, l_0, \ldots, l_m) \\ \mathbf{end}\end{array} \tag{4.3}$$

[1] The compilation of the *Circus* refinement laws mentioned in this thesis can be found in the Appendix D.

We now move to one of the biggest change in the structure of the process, with a data refinement that transforms the original state, with multiple variables, into a single binding component, b of type $BINDING$, as already presented here.

$$
\begin{array}{l}
\textbf{process}\ P \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [b : BINDING \mid b(v_0) \in T_0 \land \ldots \land inv(b(v_0), \ldots, b(l_m))] \\
\quad \bullet\ A(b(v_0), \ldots, b(v_n), b(l_0), \ldots, b(l_m)) \\
\textbf{end}
\end{array}
\tag{4.4}
$$

Now that the state is redefined concerning the bindings, we then introduce a new *Circus* action, $Memory$, that manages the interaction between the main action with the new form of access to the state components. At the same time, according to the definition B.41 [129, p. 186], the declaration part $decl$ of a process state can be also be defined as a local variable of the main action, $\mathbf{var}\ decl \bullet A$. Therefore, the declaration of the bindings in $State$ is transformed into a local variable of the main action. This step differs from [132] as we are not preserving the state invariants in the local variables. Such step, however, is justified as in the semantic model of *Circus* processes: it ignores any existing state invariants [129, p. 67][132, p. 89]. Moreover, capturing the invariants during model-checking may result in a much more complex structure and, therefore, can lead to state explosion during the refinement checks, as already discussed in Section 3.5.

$$
\begin{array}{l}
\textbf{process}\ P' \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [b : BINDING \mid b(v_0) \in T_0 \land \ldots \land inv(b(v_0), \ldots, b(l_m))] \\
\quad Memory \mathrel{\widehat=} \\
\qquad \textbf{vres}\ b : BINDING \bullet \\
\qquad\quad (\Box\, n : \mathrm{dom}\ b \bullet mget.n!b(n) \to Memory(b)) \\
\qquad\quad \Box \left(\Box\, n : \mathrm{dom}\ b \bullet \left(\begin{array}{l} mset.n?nv : (nv \in \delta(n)) \to \\ Memory(b \oplus n \mapsto nv) \end{array} \right) \right) \\
\qquad\quad \Box\, terminate \to \mathrm{Skip} \\
\quad \bullet\ \textbf{var}\ b : BINDING \bullet \\
\qquad \left(\left(\begin{array}{l} \Omega_A(A); \\ terminate \to \mathrm{Skip} \end{array} \right) [\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b) \right) \setminus MEM_I \\
\textbf{end}
\end{array}
\tag{4.5}
$$

Therefore, because of definition 1 [129, Section 3.1.4, Def. B.41, p. 43], we no longer need to include the state definition in the *Circus* process.

$$
\begin{array}{l}
\textbf{process}\ P' \mathrel{\widehat=} \\
\textbf{begin} \\
\quad Memory \mathrel{\widehat=} \\
\qquad \textbf{vres}\ b : BINDING \bullet \\
\qquad\quad (\Box\, n : \mathrm{dom}\ b \bullet mget.n!b(n) \to Memory(b)) \\
\qquad\quad \Box \left(\Box\, n : \mathrm{dom}\ b \bullet \left(\begin{array}{l} mset.n?nv : (nv \in \delta(n)) \to \\ Memory(b \oplus n \mapsto nv) \end{array} \right) \right) \\
\qquad\quad \Box\, terminate \to \mathrm{Skip} \\
\quad \bullet\ \textbf{var}\ b : BINDING \bullet \\
\qquad \left(\begin{array}{l} (\Omega_A(A)\ ;\ terminate \to \mathrm{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array} \right) \setminus MEM_I \\
\textbf{end}
\end{array}
\tag{4.6}
$$

According to the example of the $RingBuffer$, from Deliverable 24.1, the local variable b in the main action of the process P' is transformed in a replicated internal choice of all possible values of b of type $BINDING$. This step is not clearly described as a translation step in [132]. In fact, this is a missing

step between the *Circus* specification and the equivalent translated CSP_M model. We describe here this step and then will present the justification for why we decided to modify the translation scheme from this step onwards.

This translation step is justified by the definition of the **L4** for **var**, from the Unifying Theories of Programming [83, p. 70], which says that given a variable x of a certain type T, then $\mathbf{var}\, x$ is equivalent to $\sqcap\{\mathbf{var}\, x := k \mid k \in T\}$. In other words, the initial value of a declared variable x is arbitrarily non-deterministic. Thus, this can be rewritten as $\mathbf{var}\, x : T = \sqcap\, x : T$, and it is in fact how we capture the **var** construct in CSP_M.

$$
\begin{array}{l}
\mathbf{process}\, P' \mathrel{\widehat=} \\
\mathbf{begin} \\
\quad Memory \mathrel{\widehat=} \\
\qquad \mathbf{vres}\, b : BINDING \bullet \\
\qquad\quad (\Box\, n : \mathrm{dom}\ \ b \bullet mget.n!b(n) \rightarrow Memory(b)) \\
\qquad\quad \Box \left(\Box\, n : \mathrm{dom}\ \ b \bullet \left(\begin{array}{l} mset.n?nv : (nv \in \delta(n)) \rightarrow \\ Memory(b \oplus n \mapsto nv) \end{array} \right) \right) \\
\qquad\quad \Box\, terminate \rightarrow \mathrm{Skip} \\
\quad \bullet \sqcap\, b : BINDING \bullet \\
\qquad \left(\begin{array}{l} (\Omega_A(A)\ ;\ terminate \rightarrow \mathrm{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array} \right) \setminus MEM_I \\
\mathbf{end}
\end{array}
\tag{4.7}
$$

Whilst testing our tool, we identified that by translating the Step 4.7 into CSP_M, the approach works fine for specifications that uses a small number of state variables. However, it becomes unfeasible in larger specifications, as the number of generated states grows in terms of the bindings combinations of state variables and their type range of values. In large scale specifications, this approach would consume an expresive amount of time and memory for FDR4 to explore the state space.

As part of our contribution to state of the art, in order to overcome such problems, we developed a new sequence of translation steps that still preserves the semantics of *Circus* and, at the same time, optimises the computational time using FDR4. Our first step is to define another refinement step after Step 4.6, where, instead of refining the local variable b of type $BINDING$ as an internal choice of all possible values of its type, we use the refinement law 21, $crl_proc_splitting4$, where the local variable is transformed into a parameter of the *Circus* process. Such a step is in conformance with the definition of a *Circus* process [129, p. 23], where parameters may be used as local variables in the definition of the process. In Chapter 5, we describe the modifications we had to include that reflects the steps presented here in the context of CSP_M.

$$
\begin{array}{l}
\mathbf{process}\, P' \mathrel{\widehat=} b : BINDING \\
\mathbf{begin} \\
\quad Memory \mathrel{\widehat=} \\
\qquad \mathbf{vres}\, b : BINDING \bullet \\
\qquad\quad (\Box\, n : \mathrm{dom}\ \ b \bullet mget.n!b(n) \rightarrow Memory(b)) \\
\qquad\quad \Box \left(\Box\, n : \mathrm{dom}\ \ b \bullet \left(\begin{array}{l} mset.n?nv : (nv \in \delta(n)) \rightarrow \\ Memory(b \oplus n \mapsto nv) \end{array} \right) \right) \\
\qquad\quad \Box\, terminate \rightarrow \mathrm{Skip} \\
\quad \bullet \quad \left(\begin{array}{l} (\Omega_A(A)\ ;\ terminate \rightarrow \mathrm{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array} \right) \setminus MEM_I \\
\mathbf{end}
\end{array}
\tag{4.8}
$$

With this approach, instead of leaving the task of generating all possible combination of bindings for the model-checker, we produce a more concrete model, where a set of bindings b should be provided as a parameter. In other words, the initial values of the *Circus* process is therefore defined non-deterministically. We assume that the **state** of a *Circus* process is explicitly initialised in the main action, which is usually one of the verification requirements from certification authorities [65].

In order to illustrate our arguments on the effect of removing the internal choice of possible bindings, in Figure 4.1, we could start the execution of our *Circus* process P' with several bindings sets, $P'(b_1), P'(b_2), \ldots, P'(b_n)$, and, after the execution of the state initialisation, $P'.StateInit$, the next state of the process would lead to a same state $State'$ for any instance of b_n, and therefore, the execution of the process main action would be identical despite the bindings used. We stress that by disregarding the state initialisation in a specification would lead the system to unexpected behaviour.

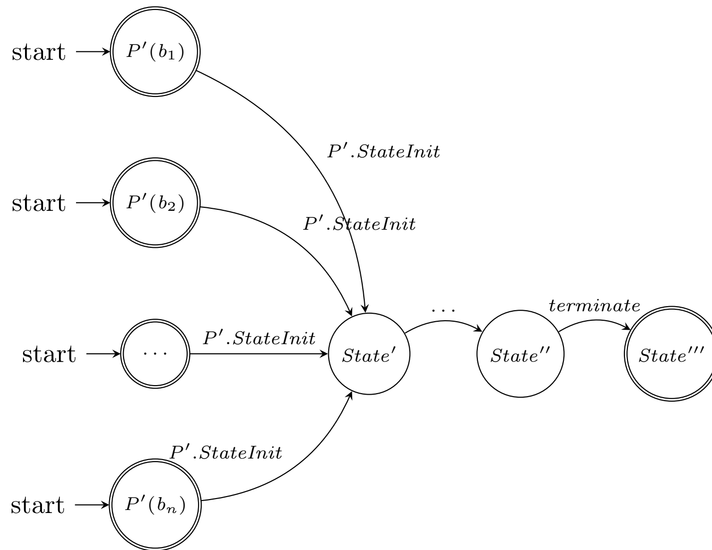


Figure 4.1: Converging transitions after the state initialisation.

In the next section we focus on the memory process, where we present the limitations found whilst testing the tool and the improvements made to that process.

4.1 Upgrading the Memory Model

With the initial version of the tool, we could take examples from D24.1 (e.g. the ring-buffer example [132, Appendix D.2, p163]) and automatically translate them and then successfully perform FDR checks. However, when we turned out attention to the haemodialysis machine [75] model, we immediately uncovered some limitations of the original translation. All of these limitations were overcome by changing the memory model only.

4.1.1 Limitation 1: Z types vs. CSP_M types

One of the issues we encountered while testing the translation rules from [132] was related to the definition of the universe type in CSP_M. In *Circus*, we abstract any type within the universe type

with [$UNIVERSE$]. However, in CSP_M, we can't use that level of abstraction and therefore, we need to produce a structure for explicitely declaring the types involved in a specification environemnt. As of now, we are not detailing the CSP_M specifities: we discuss the CSP_M environment structure and how we handle the type definitions in Section 5.1.

We anticipate that the examples used while testing our tool were enough to show that FDR does not support a polymorphic definition of $BINDING$, and therefore, we had to refine our memory model, as well as to partition the $UNIVERSE$ and $BINDING$ types into distinct sets for each type used within a specification.

$$
\begin{array}{l}
\textbf{process}\ P' \mathrel{\widehat=} b_{T_1} : BINDING_{T_1}, \ldots, b_{T_n} : BINDING_{T_n} \\
\textbf{begin} \\
\quad Memory \mathrel{\widehat=} \\
\quad\quad \textbf{vres}\ b_{T_1} : BINDING_{T_1}, \ldots, b_{T_n} : BINDING_{T_n} \bullet \\
\quad\quad\quad (\Box\, n : \mathrm{dom}\ b_{T_1} \bullet mget.n!b_{T_1}(n) \rightarrow Memory(b_{T_1}, \ldots, b_{T_n})) \\
\quad\quad\quad \Box \ldots \\
\quad\quad\quad \Box(\Box\, n : \mathrm{dom}\ b_{T_n} \bullet mget.n!b_{T_n}(n) \rightarrow Memory(b_{T_1}, \ldots, b_{T_n})) \\
\quad\quad\quad \Box \left(\Box\, n : \mathrm{dom}\ b_{T_1} \bullet \left(\begin{array}{l} mset.n?nv : (nv \in \delta(n)) \rightarrow \\ Memory((b_{T_1} \oplus \{n \mapsto nv\}), \ldots, b_{T_n}) \end{array} \right) \right) \\
\quad\quad\quad \Box \ldots \\
\quad\quad\quad \Box \left(\Box\, n : \mathrm{dom}\ b_{T_n} \bullet \left(\begin{array}{l} mset.n?nv : (nv \in \delta(n)) \rightarrow \\ Memory(b_{T_1}, \ldots, (b_{T_n} \oplus \{n \mapsto nv\})) \end{array} \right) \right) \\
\quad\quad\quad \Box\, terminate \rightarrow \mathrm{Skip} \\
\quad \bullet \left(\begin{array}{l} (\ \Omega_A(A)\ ;\ terminate \rightarrow \mathrm{Skip}\ ) \\ [\![\varnothing \mid MEM_I \mid \{b_{T_1}, \ldots, b_{T_n}\}]\!] \\ Memory(b_{T_1}, \ldots, b_{T_n}) \end{array} \right) \setminus MEM_I \\
\textbf{end}
\end{array}
\tag{4.9}
$$

The above-presented model resolved the problem with CSP_M type limitations and allowed the successful loading of the translated HD model into FDR. However, it now exposed the second limitation.

4.1.2 Limitation 2: FDR time/space explosion

We quickly discovered that we could only check small *Circus* models using this translation, with even the hand-translation of the HD model done for the original case-study being more effective. We proceeded to experiment with transformations to the memory model, justified by the *Circus* refinement laws. The final step, observing a trend, was to do more partitioning, moving to a situation were every variable gets its memory process. The supertype bindings were retained at the top-level, but the relevant binding parameterised each variable's memory process with its domain restricted to just the name of that variable. So, for example, if variable n_i has a type whose type is T, then we first define a binding b_T for that type and use it to parameterise a memory process for all variables of that type. Then we have a parallel composition of a memory process for each such variable, all synchronising on $terminate$, but interleaving all the $mget$ and $mset$ events:

$$MemoryT(b_T) \mathrel{\widehat=} [\![\{\!|\ terminate\ |\!\}]\!]\ n : \mathrm{dom}\ b_T \bullet MemoryTVar(n, \{n\} \lhd b_T) \tag{4.10}$$

Here $N \lhd \mu$ restricts the domain of map μ to set N. We then define a parameterised process that represents a single variable:

$$MemoryTVar(n, b) \mathrel{\widehat=} \left(\begin{array}{l} mget.n.b(n) \rightarrow MemoryTVar(n, b) \\ \Box\, mset.n?nv : \delta(n) \rightarrow MemoryTVar(n, b \oplus n \mapsto nv) \\ \Box\, terminate \rightarrow \mathrm{Skip} \end{array} \right) \tag{4.11}$$

Remember that δ maps a variable name to its type. The entire memory is constructed by putting the memories for each supertype in parallel, in the same as for the individual variable processes.

$$Memory(b_{T_1}, \ldots, b_{T_k}) \mathrel{\widehat=} \quad MemoryT_1(b_{T_1}) [\![\{\!| terminate |\!\}]\!] \ldots [\![\{\!| terminate |\!\}]\!] MemoryT_k(b_{T_k}) \quad (4.12)$$

This last transformation produced a marked improvement in the time and memory consumption of FDR when checking models.

$$\begin{array}{l}
\textbf{process } P' \mathrel{\widehat=} b_{T_1} : BINDING_{T_1}, \ldots, b_{T_n} : BINDING_{T_n} \\
\textbf{begin} \\
\quad MemoryTVar(n, b) \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} mget.n.b(n) \rightarrow MemoryTVar(n, b) \\ \Box\, mset.n?nv : \delta(n) \rightarrow MemoryTVar(n, b \oplus n \mapsto nv) \\ \Box\, terminate \rightarrow \text{Skip} \end{array} \right) \\
\quad MemoryT(b_T) \mathrel{\widehat=} \\
\qquad [\![\{\!| terminate |\!\}]\!]\ n : \operatorname{dom} b_T \bullet MemoryTVar(n, \{n\} \lhd b_T) \\
\quad Memory(b_{T_1}, \ldots, b_{T_k}) \mathrel{\widehat=} \left(\begin{array}{l} MemoryT_1(b_{T_1}) \\ [\![\{\!| terminate |\!\}]\!] \\ \ldots \\ [\![\{\!| terminate |\!\}]\!] \\ MemoryT_k(b_{T_k}) \end{array} \right) \\
\bullet \left(\begin{array}{l} (\ \Omega_A(A)\ ;\ \ terminate \rightarrow \text{Skip}\ ) \\ [\![\varnothing \mid MEM_I \mid \{b_{T_1}, \ldots, b_{T_n}\}]\!] \\ Memory(b_{T_1}, \ldots, b_{T_n}) \end{array} \right) \setminus MEM_I \textbf{ end}
\end{array} \quad (4.13)$$

The above definition concludes the refinement steps in order to transform a state-rich *Circus* process into a state-poor process, where the *Memory* action controls the data previously stored in the state of the process. We also presented here our proposed translation strategy as a way to overcome the limitations regarding the use of polymorphic functions in CSP_M as well as why we removed the replicated internal choice from the main action in order to reduce the state exploration in FDR.

4.2 Rewritting *Circus* Actions with Ω

We implemented the Ω functions that transform state-rich into state-poor *Circus* processes. The implementation of the Omega functions[2] in the *Circus* action level is performed using a set of recursive functions over the *Circus* actions AST. These functions are applied in order to produce a specific shape for the main action of a *Circus* process, where the reference to any state component or local variables is made using a local copy, obtained by *mget*, each of them with $v_$ prefixed to their names.

We use our *Circus* AST in Haskell, and therefore, we can create functions that match the patterns for the translation rules. Such a feature makes the implementation very straightforward for most of the *Circus* constructs.

One first example of translation rule is the prefixed action $c \rightarrow A$, which does not require or use any data from the state variables, and therefore, it only recurses over the construct.

$$\Omega_A(c \rightarrow A) \mathrel{\widehat=} c \rightarrow \Omega_A(A)$$

It is then implemented in Haskell using pattern matching, using the construct `(CSPCommAction x a)` for prefixed actions, where x is used for the channel declaration, and a

[2]The entire list of implemented functions is available for further reading in Appendix B.

is the subsequent action. In this case, we use the pattern (ChanComm c []) to represent that the channel c does not communicate any value, represented by the empty list of channel communication []. As described above, the Ω_A for such pattern recurses over a.

```
omega_CActions :: CAction -> CAction
omega_CActions (CSPCommAction (ChanComm c []) a)
  = (CSPCommAction (ChanComm c []) (omega_CActions a))
```

A slightly more complex example is that of events outputting values that involve state variable values. In such a case, we must identify the set of free variables of the expression e which will be the output of the channel c. From that set, we prefix the action with one or more *mget* events, retrieving the values of the related state variables to the expression e.

$$\begin{array}{l}\Omega_A(c.e(v_0,\ldots,v_n,l_0,\ldots,l_m) \rightarrow A) \mathrel{\widehat=} \\ \qquad mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\ \qquad mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\ \qquad c.e(vv_0,\ldots,vv_n,vl_0,\ldots,vl_m) \rightarrow \Omega'_A(A) \\ \quad where\ FV(e) = (v_0,\ldots,v_n,l_0,\ldots,l_m)\end{array}$$

Our implementation of the above definition is illustrated below. Firstly, we have to retrieve *fe*, the list of variables used in the expression e, using the function *getChanDotExpVar*[3]. Then, we obtain the list of free variables *fvars* and then the function can fall on two cases: (1) if *fvars* is empty, the function recurses on the process a; (2) otherwise, for each variable of *fvars*, a *mget* event is created in order to retrieve the value of that state variable. In *Circus2CSP*, we use the auxiliary function make_get_com which creates the *mget* actions for each element of the list of free variables – which retrieve the state variables used – *fvars*, in the action. Finally, we also rename the variables used in action a, with the help of the function rename_vars_CAction, so they now refer to the $v_$ prefixed variables, obtained using *mget*, rather than accessing the state variables directly.

```
omega_CAction (CSPCommAction (ChanComm c e) a) =
  case fvars of
      [] -> (CSPCommAction (ChanComm c e) (omega_CAction a))
       _ -> make_get_com fvars (rename_vars_CAction (CSPCommAction (ChanComm c e) (omega_prime_CAction a)
          ))
  where fe = getChanDotExpVar e
         fvars = (remdups ( concat ( map get_ZVar_st ( concat ( map varset_to_zvars  (map free_var_ZExpr
              fe))))))
```

A third example which requires a more complex implementation is the case of the use of the external choice operator. First, we have to evaluate which state variables are mentioned in each side of the operator, and for each state variable, one *mget* prefixed action should be performed right before the scope of the operator, so those values can then be used in both sides. Mainly, the Ω'_A set of action behaves just like Ω_A, however, does not produce any *mget* or *mset* before the construct.

$$\Omega_A(A_1 \Box A_2) \mathrel{\widehat=} \left(\begin{array}{l} mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\ mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\ \quad (\Omega'_A(A_1) \Box \Omega'_A(A_2)) \end{array}\right)$$

The implementation is similar to the previous example presented above, where we use to retrieve the free variables of the action in question and then, prefix the external choice with the relevant *mget* events. Finally, the Ω'_A functions are used for both sides of the external choice, meaning that the

[3]The source code of *Circus2CSP* is available online [76].

values obtained with *mget* are propagated to each side of the choice, and no further *mget* events should happen until required, as defined by the translation rules.

```
omega_CAction (CSPExtChoice ca cb)
  = make_get_com fvars (CSPExtChoice (omega_prime_CAction ca)
                                     (omega_prime_CAction cb))
  where
  fvars = remdups $ concat $ (map get_ZVar_st $ varset_to_zvars $
             free_var_CAction (CSPExtChoice ca cb))
```

Moreover, any replicated operator such as replicated sequential composition or external choice are translated into their expanded form, which will then match the respective pattern for such operator, since the Ω_A function is called again for such expanded form.

$$\Omega_A(\mathbin{;}\ x : \langle v_1, ..., v_n\rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1)\ ;\ \ldots ;\ A(v_n))$$
$$\Omega_A(\Box\, x : v_1, ..., v_n \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1) \Box \ldots \Box A(v_n))$$

Another example we present here is the translation for assignments. Such a construct is not present in the syntax of CSP_M and therefore, needs to be transformed into something else in *Circus* which can then be translated into CSP_M. We use as an example the definition of the *AInit* action of the *WakeUp* *Circus* process from Section 2.4.2, where the assignment operator is used in order to set the initial values for *sec* and *min* as 0, and *buzz* as *OFF*.

$$\Omega_A(sv_sec,\ sv_min,\ sv_buzz\ :=\ 0, 0, OFF) \mathrel{\widehat=} \left(\begin{array}{l} mset.sv_sec.0 \rightarrow \\ mset.sv_min.0 \rightarrow \\ mset.sv_buzz.OFF \rightarrow \text{Skip} \end{array}\right)$$

We can observe that no *mget* event occurred in *AInit*. It is because the values that will be assigned to the variables does not depend on other state variables. A different example that uses both *mget* and *mset* events is illustrated below. Let's translate now the action *IncSec* from the same example of Section 2.4.2:

$$\Omega_A(sec, min := (sec + 1) \bmod 3, min) \mathrel{\widehat=} \left(\begin{array}{l} mget.sv_sec?v_sv_sec : \delta(sv_sec) \rightarrow \\ mget.sv_min?v_sv_min : \delta(sv_min) \rightarrow \\ mset.sv_sec.(NAT.((v_sv_sec + 1) \bmod 3)) \rightarrow \\ mset.sv_min.v_sv_min \rightarrow \text{Skip} \end{array}\right)$$

Basically, the result transformation is a sequence of *mgets* and *msets* from the *Memory* model. We first get, *mget*, all the state variable and local variable values used in the left-hand side of the assignment, and then, using *mset*, we set the values of such variables. The most interesting translation rule, and yet the most complex, is the one that deals with actions in parallel. In *Circus* the actions A_1 and A_2 are modelled to work in parallel as $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$. The initial values of all variables are available for both actions. However, A_1 can only modify the variables mentioned in the name set ns_1. Likewise, ns_2 represents the set of permitted variables to be modified by A_2.

In D24.1, the translation uses two auxiliary actions: *MemoryMerge* and *Merge*. The former behaves like *Memory* using a copy of the bindings, except in the case of a *terminate* signal, when it propagates two other signals *mleft* and *mright* to the later, which will write the final values to *Memory*

according to the state partition.

$$
\begin{array}{l}
\Omega_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2) \mathrel{\widehat=} \\
\quad mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\
\quad mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(\ \Omega'_A(A_1)\ ;\ terminate \rightarrow \mathrm{Skip}\ ) \\
\llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\
MemoryMerge(\{v_0 \mapsto vv_0, \ldots\}, LEFT)
\end{array}\right) \setminus MEM_I \\
\llbracket \{\} \mid cs \mid \{\} \rrbracket \\
\left(\begin{array}{l}
(\ \Omega'_A(A_2)\ ;\ terminate \rightarrow \mathrm{Skip}\ ) \\
\llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\
MemoryMerge(\{v_0 \mapsto vv_0, \ldots\}, RIGHT)
\end{array}\right) \setminus MEM_I
\end{array}\right) \\
\llbracket \{\} \mid MRG_I \mid \{\} \rrbracket \\
Merge
\end{array}\right) \\
\quad \setminus MRG_I
\end{array}
$$

where

$$
\begin{array}{l}
Merge \mathrel{\widehat=} \left(\begin{array}{l}
(mleft?l \rightarrow \mathop{;}\ n : ns_1 \bullet mset.n!l(n) \rightarrow \mathrm{Skip}) \\
\interleave (mright?l \rightarrow \mathop{;}\ n : ns_2 \bullet mset.n!l(n) \rightarrow \mathrm{Skip})
\end{array}\right) \\
MemoryMerge \mathrel{\widehat=} \\
\quad \mathbf{vres}\ b : BINDING;\ s : SIDE \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\Box\ n : \mathrm{dom}\, b \bullet mget.n!b(n) \rightarrow \\
\qquad MemoryMerge(b, s)
\end{array}\right) \\
\Box \left(\begin{array}{l}
\Box\ n : \mathrm{dom}\, b \bullet mset.n?nv : (nv \in \delta(n)) \rightarrow \\
\qquad MemoryMerge(b \oplus \{n \mapsto nv\}, s)
\end{array}\right) \\
\Box terminate \rightarrow \left(\begin{array}{l}
(s = LEFT)\ \&\ mleft!b \rightarrow \mathrm{Skip} \\
\Box\ (s = RIGHT)\ \&\ mright!b \rightarrow \mathrm{Skip}
\end{array}\right)
\end{array}\right)
\end{array}
$$

In our model, we redesigned the translation rule for parallel actions. We isolate completely the local copies of initial values of the bindings with Γ_A functions instead of Ω_A, which uses local gets and sets, namely *lget* and *lset*, as an intermediate state. It will propagate the local state values from *MemoryMerge* to *Memory* should a *lterminate* signal occurs, meaning that the parallelism has finished. Hence, the new Ω_A translation rule for parallel actions is illustrated as follows.

$$
\begin{array}{l}
\Omega_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2) \mathrel{\widehat=} \\
\quad mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\
\quad mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\
\quad \left(\left(\begin{array}{l}
\left(\begin{array}{l}
(\ \Gamma'_A(A_1)\ ;\ lterminate \rightarrow \mathrm{Skip}\ ) \\
\llbracket \{\} \mid MEM_L \mid \{\} \rrbracket \\
MemoryMerge(b_TYP_1, \ldots, b_TYP_n, ns1)
\end{array}\right) \setminus MEM_L \\
\llbracket \{\} \mid cs \mid \{\} \rrbracket \\
\left(\begin{array}{l}
(\ \Gamma'_A(A_2)\ ;\ lterminate \rightarrow \mathrm{Skip}\ ) \\
\llbracket \{\} \mid MEM_L \mid \{\} \rrbracket \\
MemoryMerge(b_TYP_1, \ldots, b_TYP_n, ns2)
\end{array}\right) \setminus MEM_L
\end{array}\right)\right)
\end{array}
$$

We follow the idea of the distributed memory system and therefore, we define a distributed *MemoryMerge* action, where we first split it into several copies, one for each binding type, synchronising on *lterminate*, resulting in several $MemoryMergeTYP_{TYP_n}$ actions, one for each type TYP_n.

$$
\begin{array}{l}
MemoryMergeTYP_1(b_TYP_1, ns) \mathrel{\widehat=} \\
\quad \llbracket \{\!|\ lterminate\ |\!\}\ \rrbracket\ v : \mathrm{dom}\, b_TYP_1 \bullet MemoryMergeTYP_1\, Var(v, b_TYP_1, ns) \\
MemoryMerge(b_TYP_1, \ldots, b_TYP_n, ns) \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
MemoryMergeTYP_1(b_TYP_1, ns) \\
\llbracket \{\!|\ lterminate\ |\!\} \rrbracket \\
\ldots \\
\llbracket \{\!|\ lterminate\ |\!\} \rrbracket \\
MemoryMergeTYP_n(b_TYP_n, ns)
\end{array}\right)
\end{array}
$$

Then, a second level of distribution is made within the scope of $MemoryMergeTYP_{T_n}$, where for every single state variable v within the bindings b_TYP_1, an action $MemoryMergeTYP_n\,Var$ is available for it and may perform either *lget* and *lset*, updating locally the values for its variable, and terminates with *lterminate*, followed by a *mset* updating its value to the memory model, should such variable belongs to the name set ns. Otherwise, it skips.

$$
\begin{array}{l}
MemoryMergeT_n\,Var(v, b_TYP_n, ns) \mathrel{\widehat=} \\
\quad \left(\begin{array}{l}
lget.v.(b_TYP_1(v)) \rightarrow MemoryMergeTYP_1\,Var(v, b_TYP_1, ns) \\
\Box\, lset.v?nv : typeTYP_1(n) \rightarrow \\
\qquad MemoryMergeTYP_1\,Var(v, b_TYP_1 \oplus \{v \mapsto nv\}), ns)) \\
\Box\, lterminate \rightarrow \left(\begin{array}{l}
\mathbf{if}\ v \in ns \longrightarrow (mset.v.(b_TYP_1(v)) \rightarrow \mathrm{Skip}) \\
\quad [\!]\, v \notin ns \longrightarrow \mathrm{Skip} \\
\mathbf{fi}
\end{array}\right)
\end{array}\right)
\end{array}
$$

Regarding the *Merge* action, it was literally merged into the $MemoryMergeTYP_n\,Var$ action, and therefore, neither *mleft* nor *mright* are any longer necessary: because of the distributed model, there is no need to model a replicated sequential composition of *mset* for every variable n in ns, once it is done locally in every action $MemoryMergeTYP_n\,Var$.

Until now, we presented the elements used in the translation strategy in order to reshape a *Circus* specification in such a way it can then be rendered in CSP_M using the Υ functions. In the next section, we present details on the CSP_M environment, as well as how we implemented the Υ functions.

Chapter 5

Translating *Circus* to CSP_M with Υ

In this section we present how we translate the transformed $Circus_{CSP}$ into CSP_M. We first present the original structure of the file [132], then we present how the improvements presented in previous sections affected such structure. Finally, we show some examples of the Upsilon (Υ) translation rules for both *Circus* processes and actions.

5.1 The CSP_M type environment for the *Memory* model

We introduce here the structure defined in Deliverable 24.1 [132], and then we present the modifications to the CSP_M environment that reflects the improvements made during our research in order to overcome the limitations of the original translation strategy, as presented in Section 4.1.

The use of the *UNIVERSE* type, the CSP_M `subtype` feature, and the function `type` written in CSP_M to map a name to its specific type worked fine if all the types in *UNIVERSE* where a sub-type of one supertype. In the D24.1 examples, all types were sub-types of the natural numbers. However, in the haemodialysis model [75], we had a mixture of types, some natural sub-types, but others being enumerations. The translations of these failed to type check in FDR because its type system is not powerful enough. The function `type` maps variables to their types in the bindings, which requires returning different types, leading to a type error in FDR. Enumerations are isomorphic to subsets of the naturals, and so we could have re-written those types, but we felt that it would be preferable if the translator could somehow handle this situation itself.

As a way of illustrating the whole structure, we use the *WakeUp* model from Section 2.4.2, which can be found in the Appendix H.4.1 along with some reasoning about that model. The only difference between the *WakeUp* model to the *Chronometer* model, from Section 3.5, is that we introduce the feature of an alarm for a defined time. Therefore, we introduce a type *ALARM* which can either be *ON* or *OFF*, depending on the behaviour of the chronometer.

Firstly, we introduce how *UNIVERSE*, *NAME*, `type`, `tag`, and `value` would be defined following [132]. We have two types in our example: the "nametype" *ALARM*, which can be *ON* and *OFF*; and the *RANGE* set, a subset of the natural numbers, ranging from 0 up to 5. The translated specification in CSP_M uses the *UNIVERSE* type that includes, for example, the type *RANGE*, indicated by using using a CSP tag $RAN.RANGE$, which is incorporated as a subtype of *UNIVERSE*. Any value of *RANGE* will now be available within *UNIVERSE* as $\{RAN.0, RAN.1, RAN.2, RAN.4, RAN.5\}$.

```
RANGE = {0..5}
datatype ALARM = ON | OFF
datatype UNIVERSE = RAN.RANGE | ALA.ALARM
```

The next step is to define subtypes of the universe type: we prefix each subtype with $U_$ followed by the first three letters of the type name, in capital letters, *e.g.* U_RAN. Each subtype is defined with a tag, and its type name, *e.g.* $RAN.RANGE$.

```
subtype U_RAN = RAN.RANGE
subtype U_ALA = ALA.ALARM
```

Here we propose a solution where the whole construction of bindings and auxiliary functions, along with how the communication in the *Memory* process is differentiated according to each type used in the specification. We modified the universal type approach from [132], to handle a mixed-type universe.

Instead of directly defining the three untypeable CSP_M functions, `value`, `type`, and `tag`, we define variants of each function for each type within the specification. For instance, for the $RANGE$ type, we define functions `valueRAN`, `typeRAN` and `tagRAN`. Further in this section, we will show that the use of the `tag` function becomes redundant for our approach.

```
valueRAN(RAN.v) = v
valueALA(ALA.v) = v

typeRAN(x) = U_RAN
typeALA(x) = U_ALA

tagRAN(x) = RAN
tagALA(x) = ALA
```

We also define subtypes for each of the $NAME$ types, and thus, its subtypes $NAME_RAN$ and $NAME_ALA$.

```
datatype NAME = sv_sec | sv_min | sv_buzz
subtype NAME_RAN = sv_sec | sv_min
subtype NAME_ALA = sv_buzz
```

Next, we define a series of of bindings for each type, instead of a universal binding mapping. For our example, we will have either $NAMES_VALUES_RAN$ and $NAMES_VALUES_ALA$, as well as its related $BINDINGS$, $BINDINGS_RAN$ and $BINDINGS_ALA$. In practice, this solution is beneficial whilst using FDR, as the checker doesn't need to build mappings for a name other than its own type. For instance, we will never have a match between a *min* variable and the value ON or OFF.

```
NAMES_VALUES_RAN = seq({seq({(n,v) | v <- typeRAN(n)}) | n <- NAME_RAN})
BINDINGS_RAN = {set(b) | b <- set(distCartProd(NAMES_VALUES_RAN))}
```

We also have to modify the definition of *Memory* in order to differentiate the bindings for every type defined in the specification. Moreover, as we now have multiple definitions of the `type` function, we need to produce instances of *mset* and *mget* for every defined type.

Firstly, we redefine the parameter of the function $Memory(binds)$, as it might have several bindings. As a convention, we define the bindings parameters as a prefix $b_$, followed by the tag name of the type. From our example, we define $Memory(b_RAN, b_ALA)$ with the bindings for $RANGE$ and $ALARM$. We illustrate the definitions for the type U_RAN and omit the definitions for type U_ALA as they are similar to the ones for U_RAN. Thus, we translate the distributed model of the *Memory* action, as introduced in Section 4.1.2.

```
MemoryRANVar(n,b_RAN) =
    mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
    [] mset.n?nv:RANeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv))
    [] terminate -> SKIP
MemoryRAN(b_RAN) =
    ( [| {| terminate |} |] n : dom(b_RAN)
              @ MemoryRANVar(n,ddres(n,b_RAN)) )
Memory(b_RAN, b_ALA) =
    MemoryALA(b_ALA) [| {| terminate |} |] MemoryRAN(b_RAN)
```

Likewise, the definitions for *MemoryMerge* in CSP_M are presented below. We make use of several auxiliar CSP_M definitions[1] such as *ddres*, which represents the domain restriction function from Z. Moreover the `over(b_RAN,n,nv)` function represents the expression $b_RAN \oplus \{n \mapsto nv\}$. In this example, we also ommit the definitions for U_ALA since they are similar to the ones for U_RAN.

```
MemoryMergeRANVar(n,b_RAN,ns) =
   lget.n.apply(b_RAN,n) -> MemoryMergeRANVar(n,b_RAN,ns)
   [] lset.n?nv:typeRAN(n) -> MemoryMergeRANVar(n,over(b_RAN,n,nv),ns)
   [] lterminate -> ( ; bd : <b_RAN>
                     @ ; n : <y | y <- ns,member(y,dom(bd))>
                         @  mset.n.apply(bd,n) -> SKIP )
MemoryMergeRAN(b_RAN,ns) =
  ( [| {| lterminate |} |] n : dom(b_RAN)
      @ MemoryMergeRANVar(n,ddres(n,b_RAN),ns) )
MemoryMerge(b_RAN,b_ALA,ns) =
  ( MemoryMergeALA(b_ALA,ns) [| {| lterminate |} |] MemoryMergeRAN(b_RAN,ns) )
```

5.2 Generating Bindings

As one of our contributions, our tool is capable of generating a simple set of bindings for each type used within the specification. *Circus2CSP* reads some of the types used within a specification and determines a binding set with the minimum value within a range of values, or the first value within a free type definition. Any other complex type definition is left as a task for the user to define it, and we point it as `DO_IT_MANUALLY`. Therefore, once type-checking with FDR, it will point exactly which binding needs to be defined manually.

We illustrate below the example of the bindings defined for the *Chronometer* model from Section 2.4.2, where we use the value 0 of type $RANGE$ for defining the values for minutes and seconds. Moreover, the buzzer of the alarm is defined as OFF.

```
b_RAN1 = {(sv_sec, RAN.0),(sv_min, RAN.0)}
b_ALA1 = {(sv_buzz, ALA.ON)}
```

[1] Two auxiliary CSP_M files are provided with the translator: function_aux.csp and sequence_aux.csp. These are derived from the Deliverable 24.1, with new other functions added.

We plan to extend the bindings generator in order to produce a mechanism capable of identifying a meaningful set of values enough for model-checking a specification without needing to explore every single value within a range. This is part of our future work and is detailed in Section 9.2

We should clarify that our approach to producing parametrised processes is not an attempt to use the bindings *data-independendly* [140, p. 453]. That is solving a different problem, namely finding a finite size of a type that is suitable to demonstrate the correctness for any finite or even infinite size of such type.

In the next section, we introduce the function Υ_P, used in order to render the structure of *Circus* processes in CSP_M.

5.3 Mapping *Circus* Processes - Υ_P

Note that the definition of both *Memory* and *MemoryMerge* actions are enclosed inside a `let-within` statement. Although, we omit the definitions of *MemoryMerge* as it is not used in the example above.

```
Proc(b_RAN,b_ALA) =
  let MemoryRANVar(n,b_RAN) = ...
      MemoryRAN(b_RAN) = ...
      MemoryALAVar(n,b_ALA) = ...
      MemoryALA(b_ALA) = ...
      Memory(b_RAN, b_ALA) =
         MemoryALA(b_ALA) [| {| terminate |} |] MemoryRAN(b_RAN)
      ...
      MemoryMerge(b_RAN, b_ALA) =
         MemoryMergeALA(b_ALA) [| {| lterminate |} |] MemoryMergeRAN(b_RAN)
  within ( ( ( MA ; terminate -> SKIP ) [| MEMI |] Memory(b_RAN, b_ALA)) \ MEMI )
```

The content of `MA` in the above extract refers to the main action of the process and it is translated into CSP_M using the Υ_A functions which will be presented in the next section.

5.4 Mapping *Circus* Actions - Υ_A

The translation from $Circus_{CSP}$ into CSP_M is basically the generation of auxiliary types from Z constructs, along with channel and processes, according to the resulting specification from the Ω functions transformation. Such a translation is made using the Υ functions, which output the CSP_M text directly into a file.

$$\Upsilon_A(c.v \to A) \mathrel{\widehat=} \texttt{c.v -> } \Upsilon_A(A)$$

During our research we identified some inconsistency in the definition of the Υ functions. We noticed that result of the translation strategy, illustrated by the *RingBuffer* [132, p. 163] example, differs from what one would obtain by using the translation rules presented in [132, p. 156]. For instance, looking at the *RingBuffer* in CSP_M, in [132, p. 170], when communicating through a channel *mget*, a state variable v of a process *RingBuffer*, translated as *RingCell_v* should obtain the value *vRingCell_v* and should state that its *type* should match with the type of *RingBuffer_v*. As an example, a channel communication like $mget.RingCell_v?vRingCell_v : \delta(RingCell_v)$ would not be captured by the two cases below.

$$\Upsilon_A(c?x : P \to A) \mathrel{\widehat=} \texttt{c?x :\{x | x <- } \delta(c)\texttt{, } \Upsilon_{\mathbb{B}}(P(x))\texttt{->} \ \Upsilon_A(A)$$
$$\Upsilon_A(c?x \to A) \mathrel{\widehat=} \texttt{c?x -> } \Upsilon_A(A)$$

Thus, there are not enough cases for the Υ function to cover all the data communication to and from *Memory*. Moreover, the two rules above do not mention the use of any of the auxiliary functions, `type`, `tag` and `value`, used in the *RingBuffer* example on [132, Appendix D4, p. 166].

It is part of our contribution to include new rules that support our running *WakeUp* example. A possible definition of a Υ function that covers the *mget* and *mset* communication, respectively, is provided below. We translate the δ function by first splitting it according to the types it returns, and create type-specific versions by adding a suffix XYZ —the type of x— to the `type`, `tag` and `value` functions.

$$\Upsilon_A(mget.x?v_x : \delta(x) \to A) \mathrel{\widehat{=}} \texttt{mget.x?v_x:(typeTYP(x)) -> } \Upsilon_A(A)$$
$$\Upsilon_A(mset.x.v_x \to A) \mathrel{\widehat{=}} \texttt{mset.x.(TYP.valueTYP(v_x)) -> } \Upsilon_A(A)$$

From our example, the *WakeUp* process, the *mset* communication is then translated to the following extract, where the auxiliary functions are renamed suffixed by RAN, the *tag* of *sv_sec*.

```
1    ... -> mset.sv_sec.(RAN.valueRAN(v_sv_sec)) -> SKIP
```

Similarly to process and action refinement, both the Ω and Υ functions are applied not only at the action level but also at the *Circus* process level. The implementation procedure is similar to that above and technical details are described in Section 5.3 of the Deliverable report D24.1 [132].

5.5 The CSP_M version of the *WakeUp* process

Moving to the main content of the specification, we use the example alarm clock, *WakeUp* process, where we incorporate the approach of separate bindings for each type. In the code extract below, we show the process structure around the equivalent code of the *Circus* main action of the *WakeUp* process.

```
1   WakeUp(b_RAN,b_ALA) =
      let MemoryRANVar(n,b_RAN) =
3             mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
              [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv))
5             [] terminate -> SKIP
          MemoryRAN(b_RAN) =
7             [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN)
          Memory(b_RAN,b_ALA) = MemoryALA(b_ALA) [| {| terminate |} |] MemoryRAN(b_RAN)
9        ...
      within ( ( (
11      mset.sv_sec.(RAN.0) -> mset.sv_min.(RAN.0) -> mset.sv_buzz.(ALA.OFF) ->
        ( let X =
13        mget.sv_buzz?v_sv_buzz:(typeALA(sv_buzz)) ->
          mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
15        mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
          ( ( ( (
17          tick ->
            mset.sv_sec.(RAN.(valueRAN(v_sv_sec) + 1) % 3) ->
19          mset.sv_min.(RAN.valueRAN(v_sv_min)) ->
            mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
21          mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
            ( ( (valueRAN(v_sv_sec) == 0) &
23            mset.sv_min.(RAN.(valueRAN(v_sv_min) + 1) % 3) ->
              mset.sv_sec.(RAN.valueRAN(v_sv_sec)) -> SKIP )
25            [] ( (valueRAN(v_sv_sec) != 0) &  SKIP ))
              [] ( (valueRAN(v_sv_min) == 1) &
27                    radioOn -> mset.sv_buzz.(ALA.ON) -> SKIP ))
              [] time -> out.(valueRAN(v_sv_min),valueRAN(v_sv_sec)) -> SKIP)
29            [] snooze -> mset.sv_buzz.(ALA.OFF) -> SKIP); X )
            within X );
31        terminate -> SKIP )
          [| MEMI |] Memory(b_RAN,b_ALA)) \ MEMI )
```

We can identify the original parts of the *Circus* specification in the CSP_M code above. For instance, the content from line 11 is the equivalent translation of the *AInit* *Circus* action. Moreover, lines 23, 24 and 25 refers to the *IncMin* action.

With this revised approach, we noticed that the `tag` function is no longer required. The main reason for that is due to the fact that we "carry" the subtype for each state variable in its own name. We redesigned the whole Jaza/*Circus2CSP* framework to support the inclusion of the type of a variable in its definition: `ZVar (n,d)`. The former definition of a `ZVar` consisted of a pair containing the variable name, n, and any decoration element, d. However, because of our findings regarding the multiple types issues in the original translation scheme [132], we decided to incorporate the variable tag, as defined in the *UNIVERSE* type within the `ZVar` definition. Therefore, we updated `ZVar (n,d)` to `ZVar (n,d,t)`, where t is the tag. For instance, the a variable sv_n of type *NatValue* is represented in Haskell as `ZVar (''sv_n'',[],''NAT'')`.

Because of the inclusion of the tag of a type in the variable definition in Haskell, we no longer need to apply the function *tag* and expect FDR to evaluate the result. Now, from our example above, in line 29, we write directly $(ALA.ON)$ instead of using $(tagALA(sv_buzz).ON)$ whilst writing the *mset* synchronisation.

In this section we presented how *Circus2CSP* renders the translated specification in CSP_M using Υ functions. We detailed the improvements made to the original translation scheme when dealing with multiple types in a same specification.

Chapter 6

A quick overview of *Circus2CSP*

In this section, we present an overall picture of how to use our tool *Circus2CSP*[1], and how it interacts with FDR4. The user can interact with the tool through a terminal-like interface, using a *REPL* (Read-Eval-Print-Loop). It provides the user with a `help` command which lists all commands available and how to use each of them. The initial screen of *Circus2CSP* is illustrated in Figure 6.1. It displays to the folder in which will look for *Circus* specifications written in LaTeX as well as the destination folder for the CSP_M files and reports from the tests performed using FDR4 in our tool.

```
Welcome to Circus2CSP translator, version X.X.X YYY 2018
Author: Artur Oliveira Gomes (gomesa@tcd.ie)
This is based on JAZA (Just Another Z Animator), see below.
Copyright(C) 1999-2005 Mark Utting (marku@cs.waikato.ac.nz).
Jaza comes with ABSOLUTELY NO WARRANTY (see file COPYING).
This is free software, and you are welcome to redistribute
it under certain conditions (see file COPYING).

Type 'help' to see the available commands.

Src. path: source/directory/
Dst. path: destination/directory/
Circus2CSP>
```

Figure 6.1: *Circus2CSP* initial screen

6.1 Commands for the translation to CSP_M

Whilst listing the available commands, as shown in Fig. 6.2 and Fig. 6.3, the `help` will display two sections: (1) a list of the commands used for the translation tool and (2) a list of commands for using the built-in assertion generator.

The list of commands for the translation section is detailed below.

"list" displays all files of the source directory

"load" loads the file containing the *Circus* specification, which should be a LaTeX file but the user must not provide the file extension `.txt`.

"reload" reloads the already opened file (no need to provide file name again)

[1] The tool can be downloaded from our repository `https://bitbucket.org/circusmodelcheck/circus2csp/commits/tag/ThesisDay`

"conv" one-command for performing: (1) translation of Z schemas (if any); (2) automatic *Circus* refinement calculus; (3) omega transformations (Ω); and (4) upsilon transformation (*Circus* to CSP_M) (Υ)

"reconv" repeats "conv"

"reset" cleans the buffer of loaded files

"quit" closes the application

```
Circus2CSP> help
Available commands for Circus2CSP:
    help                        Display this message
    list                        List the files in the current directory.
    quit                        Exit the animator
    reset                       Reset the whole specification
    load filename               Load a Circus spec from a file (do not include ".tex")
    reload                      Re-load Circus spec from current file
    conv filename               'load filename; omega; tocsp'.
    convp filename              perform conv but with mget_var instead of mget.var
    reconv                      repeat 'conv' on previous file
    reconvp                     repeat 'convp' on previous file
    % comment                   (Ignored)
    ...
```

Figure 6.2: *Circus2CSP* help menu

6.2 Injecting CSP_M in the Source Files

One interesting feature we decided to include in our tool is the possibility of injecting pieces of CSP_M directly into the LaTeX source files. We can specify a system in *Circus* and also include the assertions which will be directly exported into the resulting file containing the CSP_M translated specification.

```
\begin{assert}
  "assert SPEC [FD= IMP"
  \also "assert SPEC :[deadlock free [FD]]"
\end{assert}
```

The code must be enclosed within the environment `assert`, which is identified by *Circus2CSP* as text content that should be included in the resulting file. As illustrated above, for each assertion, the user should insert a new line and quote the entire string, and in case more than an assertion is required, the macro `\also` should be included between every two assertions. It is also possible to include other CSP_M definitions such as processes, channels and auxiliary functions when added to the LaTeX in a `assert` environment.

6.3 Integrating FDR4 with *Circus2CSP*

Model checking through FDR4 allows the user to perform a wide range of analysis of the specifications, such as refinement checks, deadlock and livelock freedom, and termination. Verifying the assertions about CSP specifications in FDR requires a considerable amount of time when we analyse more complex systems. On top of the wide range of analysis that we can perform, another useful feature available in FDR4 is to be able to animate the specification using *Probe*.

We were able to integrate *Circus2CSP* with the command line interface of FDR4, and therefore, we cannot only translate *Circus* into CSP_M but perform all possible check using FDR straight from our tool. The first two commands we introduce now, as listed in Fig. 6.3, are: `procs` and `runfdr`. The former, `procs`, lists all the CSP_M processes available, whereas the latter executes FDR4 and checks any assertion included in the LaTeX file using the environment `assertion`, as detailed in Section 6.2.

As part of our contribution, we developed an automatic generator of assertions for FDR4. Our tool is capable of creating and executing assertions regarding the currently loaded specification. It is executed using the command `assert` which provides results from refinement checks and for deadlock freedom, divergence, and determinism check, depending on the user needs.

```
Available commands for FDR4:
The parameter 'model' where m can be [T,F,FD]
    procs                       list all available processes
    runfdr                      Run FDR4 in command line mode
    assert ref spec impl        assert spec [FD= impl
    assert ref spec impl model  assert spec [m= impl
    assert refall               perform batch refinement for all processes available
    assert refall model         perform refall using a given model
    assert dl spec              checks spec for dls
    assert dl spec model        checks spec for dls using a given model
    assert dlall                perform batch dl check for all processes available
    assert dlall model          perform 'dlall' using a given mode
    assert div spec             checks spec for div
    assert div spec model       checks spec for div using a given mode
    assert divsall              perform batch div check for all processes available
    assert divsall model        perform 'divsall'  using a given model
    assert det spec             checks if the spec is deterministic
    assert det spec model       checks if the spec is deterministic using a given model
    assert detall               perform batch det check for all processes available
    assert detall model         perform 'detall'  using a given model
    assert jumbo                perform all batches available (may take some time)
```

Figure 6.3: *Circus2CSP* help menu - continuation of Fig. 6.2

For each assertion check, the user can either select a specific check, as well as selecting which process or processes and the model to be used, which can be (T) traces, (F) failures, and (FD) failures-divergence model. Moreover, the user has the option to tell the tool to perform a specific check to all available processes or to perform any combination of refinement between the available processes. Finally, the command `assert jumbo` perform all checks available by *Circus2CSP* all at once.

6.4 Outputs provided by *Circus2CSP*

In our tool, we provide the results of both translation and assertion checks in FDR4 through separated outputs.

1. `filename.csp` – Provides the translation to CSP_M of the *Circus*specification contained in `filename.tex`.

2. `filename.hc` – Contains the Haskell AST representation of the translated *Circus* specification[2].

3. `filename.checks.csp` – Uses a copy of `filename.csp` in order to perform the assertion checks.

[2]Such file may be useful for debugging and reporting issues to the authors.

4. `filename.report.txt` – Provides all results from the assertion checks of a given `filename.tex`[3].

6.5 Final Considerations

The development of *Circus2CSP* was conceived with the idea of reusing existing tools and filling gaps in order to achieve a way of model-checking *Circus* specifications using FDR. We reused Jaza, written in Haskell, rather than using the CZT framework, since we considered that the Java code of CZT was much more complicated to understand and to start developing from it. However, the Haskell code for Jaza was much more straightforward to read and include new features on top of the existing code.

Our attempt to integrate *Circus2CSP* with FDR is limited to the interface available by FDR. We integrated our tool with FDR using the executable command "refines", as part of FDR's package, and manually parse the output resulted from its execution. A first restriction is that we can not list the process names using the command-line, which is available in the GUI interface. Therefore, we implemented our name listing command in *Circus2CSP*. However, we are restricted to the list of the translated processes from *Circus* and cannot list the CSP_M content of the environment `assert` in LaTeX, since to this day, the parser recognises the content declared in that environment as pure text, which will be directly included into the CSP_M files. We are aware that we could overcome such limitation if we had integrated the CSP_M Haskell parser [4] in our tool. It would be helpful in order to be able to list all the processes, including the ones in the `assert` environment.

Yet another limitation for the integration with FDR and the use of CSP_M as a target language is that FDR provides counterexamples in CSP_M that would be difficult to interpret regarding the state properties of *Circus* processes. However, as mentioned earlier in this thesis, our focus here is to model-check the behavior of the system, and just like we do not provide a translation for preconditions and invariants, the changes of the state of a *Circus* process are also not analysed with our tool.

In the next section, we evaluate several *Circus* specification examples using *Circus2CSP*, comparing with other methodologies from the literature.

[3] As of now, the tool will provide only the verdict regarding the assertion, in a simplified way: *Passed* or *Failed*. For future work, we will include an extended output system, which will be able to provide details regarding the state space exploration, counterexamples, as well as to generate LaTeX tables for displaying the results for refinements of the combination between all of the Processes.

[4] More about the CSP_M parser can be found at `https://github.com/tomgr/libcspm`

Chapter 7

Using *Circus2CSP* and FDR4

In this chapter, we present a compilation of *Circus* examples from the literature, and for each of them, we use *Circus2CSP* in order to obtain the CSP_M code. Then we compare, whenever possible, the results obtained between our approach and the ones presented in the literature. We first introduce rather small and simple examples, which shows interesting features of *Circus*. We also illustrate with those examples, how the Ω transformations combined with the *Circus* Refinement Laws can solve problems previously mentioned in the literature.

As we move along the chapter, we introduce more complex examples and finish this with the Haemodialysis case study, comparing the results obtained since the beginning of our research, until the conclusion of *Circus2CSP*.

7.1 Simple *Circus* Examples

Our first examples, from Example 1 to Example 6 were originally presented by Beg [15, Section 9.3, p.93-99] in his PhD thesis, where *Circus* was translated into CSP_M using a different strategy, where the process state is preserved through the use of parameters, and consecutive action calls with updated values for the state variables.

Example 1 – Lift Process

This first example shows the specification of a lift, which controls the direction of the lift, as well as the doors opening and closing.

$$
\begin{array}{l}
NatValue == 0 \mathrel{..} 5 \\
DoorState ::= opened \mid closed \\
\mathbf{channel}\ up, down, open, close
\end{array}
$$

The *Circus* process *Lift* has two state variables: *floor* captures the floor level, and *doorState* describes when the door is either *opened* or *closed*. It behaviour starts with the *InitLift* action, setting the initial values for the state variables. Then, the *Run* action is called in an infinite loop, offering the options to go *up* or *down*, as well as to *open* or *close* the doors, depending on the conditions defined

for each action.

$$
\begin{array}{l}
\textbf{process}\ Lift \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ LiftState \mathrel{\widehat=} [floor : NatValue;\ doorState : DoorState] \\
\quad InitLift \mathrel{\widehat=} (floor := 0)\ ;\ (doorState := closed) \\
\quad Run \mathrel{\widehat=} \left(\begin{array}{l}
\big(floor < 5 \land doorState = closed\big)\ \&\ up \rightarrow (floor := floor + 1) \\
\Box \\
\big(floor > 0 \land doorState = closed\big)\ \&\ down \rightarrow (floor := (floor - 1)) \\
\Box \\
\big(doorState = closed\big)\ \&\ open \rightarrow (doorState := opened) \\
\Box \\
\big(doorState = opened\big)\ \&\ (close \rightarrow (doorState := closed))
\end{array}\right) \\
\quad \bullet\ InitLift\ ;\ (\mu X \bullet (Run\ ;\ X)) \\
\textbf{end}
\end{array}
$$

Then we use *Circus2CSP* for translating the *Lift* specification into CSP_M, resulting in the following code.

```
Lift(b_NAT,b_DOO) =
  let
     MemoryNATVar(n,b_NAT) =
        ( ( mget.n.apply(b_NAT,n) -> MemoryNATVar(n,b_NAT)
            [] mset.n?nv:typeNAT(n) -> MemoryNATVar(n,over(b_NAT,n,nv)))
            [] terminate -> SKIP)
     MemoryDOOVar(n,b_DOO) = ..
     MemoryNAT(b_NAT) = ( [| {| terminate |} |] n : dom(b_NAT) @ MemoryNATVar(n,b_NAT) )
     MemoryDOO(b_DOO) = ...
     Memory(b_NAT,b_DOO) = ( MemoryDOO(b_DOO) [| {| terminate |} |] MemoryNAT(b_NAT) )
  within ( ( (
    mset.sv_floor.(NAT.0) -> mset.sv_doorState.(DOO.closed) ->
    ( let X = mget.sv_doorState?v_sv_doorState:(typeDOO(sv_doorState)) ->
              mget.sv_floor?v_sv_floor:(typeNAT(sv_floor)) ->
              ( ( ( ( (
              ((valueNAT(v_sv_floor) < 5) and (valueDOO(v_sv_doorState) == closed))
                 & up -> mset.sv_floor.(NAT.(valueNAT(v_sv_floor) + 1)) -> SKIP )
              [] ( ((valueNAT(v_sv_floor) > 0) and (valueDOO(v_sv_doorState) == closed))
                 & down -> mset.sv_floor.(NAT.(valueNAT(v_sv_floor) - 1)) -> SKIP ))
              [] ( (valueDOO(v_sv_doorState) == closed)
                 & open -> mset.sv_doorState.(DOO.opened) -> SKIP ))
              [] ( (valueDOO(v_sv_doorState) == opened)
                 & close -> mset.sv_doorState.(DOO.closed) -> SKIP ));
      X ) within X );
      terminate -> SKIP )
      [| MEMI |] Memory(b_NAT,b_DOO))\MEMI )
```

Then, from the translation from Beg's tool,

```
INITLIFT(doorState,floor) = LIFT(closed,0)
LIFT(doorState,floor) =
      ((floor < 5 and doorState == closed)) &  (up -> LIFT(doorState,floor + 1))
      []
      ((floor > 0 and doorState == closed)) &  (down -> LIFT(doorState,floor - 1))
      []
      (doorState == closed) &  (open -> LIFT(opened,floor))
      []
      (doorState == opened) &  (close -> LIFT(closed,floor))
```

We compare both models by defining *ArshadLift* and *CTOCLift* processes, with the same binding values, and check for the refinement between them, which turns out to be equivalent to each other.

```
ArshadLift = INITLIFT(valueDOO(apply(b_DOO1,sv_doorState)),
                      valueNAT(apply(b_NAT1,sv_floor)))
CTOCLift = Lift(b_NAT1,b_DOO1)
assert ArshadLift [FD= CTOCLift -- Passed
assert CTOCLift [FD= ArshadLift -- Passed
```

Example 2 – Simple Sequential Chain of Calls in Initialiser

In this example, assignments are combined with channel communication, in a sequential compostition. The refinement calculator is used in order to transform any sequential composition into a single sequence of communication, since assignments are transformed into *mget* and *mset* events. For all the examples from now on, we restrict the range of values of an assignment using the mod operator.

$$
\begin{array}{l}
\textbf{channel}\ a, b, c \\
\textbf{process}\ ProcEx1 \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ Ex1State \mathrel{\widehat=} [x, y : NatValue] \\
\quad InitProcEx1 \mathrel{\widehat=} (x := 1) \mathbin{;} (y := 1) \\
\quad ActionA \mathrel{\widehat=} (x := 0) \mathbin{;} (a \rightarrow \mathrm{Skip}) \\
\quad ActionB \mathrel{\widehat=} (y := x + 1 \bmod 5) \mathbin{;} (b \rightarrow \mathrm{Skip}) \\
\quad ActionC \mathrel{\widehat=} c \rightarrow \mathrm{Skip} \\
\quad \bullet\ ActionC \mathbin{;} ActionA \mathbin{;} ActionB \\
\textbf{end}
\end{array}
$$

Then we use *Circus2CSP* for translating the $ProcEx1$ specification into CSP_M, resulting in the following code.

```
1   ProcEx1(b_NAT) =
      let
3       MemoryNATVar(n,b_NAT) = ...
        MemoryNAT(b_NAT) = ...
5       Memory(b_NAT) = MemoryNAT(b_NAT)
      within ( ( (
7       c -> -- ActionC
        mset.sv_x.(NAT.0) -> a -> -- ActionA
9       mget.sv_x?v_sv_x:(typeNAT(sv_x)) -> -- ActionB
        mset.sv_y.(NAT.((valueNAT(v_sv_x) + 1)%5)) -> b -> SKIP; -- ActionB
11      terminate -> SKIP )
          [| MEMI |] Memory(b_NAT))\MEMI )
```

The translation provided by Beg is:

```
    ACTION_C(x,y) = c -> ACTION_A(x,y)
2   ACTION_A(x,y) = a -> ACTION_B(0,y)
    ACTION_B(x,y) = b -> SKIP
```

Then we define the two CSP_M processes $ArshadProEx1$ and $CTOCProcEx1$, and check the refinement between them, proving its equivalence.

```
1   ArshadProEx1 = ACTION_C(valueNAT(apply(b_NAT1,sv_x)),
                            valueNAT(apply(b_NAT1,sv_y)))
3   CTOCProcEx1 = ProcEx1(b_NAT1)
    assert ArshadProEx1 [FD= CTOCProcEx1 -- Passed
5   assert CTOCProcEx1 [FD= ArshadProEx1 -- Passed
```

Example 3 – Main Action having Internal Choice

We introduce now an example which uses the sequential composition operator and the internal choice operator.

$$
\begin{array}{l}
\textbf{process}\ ProcEx2 \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ Ex2State \mathrel{\widehat=} [x, y : NatValue] \\
\quad InitProcEx2 \mathrel{\widehat=} (x := 1) \mathbin{;} (y := 1) \\
\quad ActionA \mathrel{\widehat=} (x := 0) \mathbin{;} (a \rightarrow \mathrm{Skip}) \\
\quad ActionB \mathrel{\widehat=} (y := x + 1 \bmod 5) \mathbin{;} (b \rightarrow \mathrm{Skip}) \\
\quad ActionC \mathrel{\widehat=} c \rightarrow \mathrm{Skip} \\
\bullet\ ActionA \mathbin{;} (ActionB \sqcap ActionC) \\
\textbf{end}
\end{array}
$$

We expect the refinement calculator to apply the distributivity of sequential composition over internal choice: $A\,;\ (B \sqcap C) = (A\,;\ B) \sqcap (A\,;\ C)$. Moreover, we note that because the main action does not call $InitProcEx2$, its behaviour is not translated into CSP_M. Therefore, the CSP_M code of $ProcEx2$ is:

```
1  ProcEx2(b_NAT) =
     let
3       MemoryNATVar(n,b_NAT) = ...
        MemoryNAT(b_NAT) = ...
5       Memory(b_NAT) = MemoryNAT(b_NAT)
     within ( ( ( (
7      mset.sv_x.(NAT.0) ->
       a ->
9      mget.sv_x?v_sv_x:(typeNAT(sv_x)) ->
       mset.sv_y.(NAT.((valueNAT(v_sv_x) + 1)%5)) ->
11     b -> SKIP
       |~|
13     mset.sv_x.(NAT.0) -> a -> c -> SKIP );
     terminate -> SKIP )
15   [| MEMI |] Memory(b_NAT))\MEMI )
```

Because the $mget$, $mset$ and $terminate$ events are hidden, $MEMI$, we could think of the behavior of $ProcEx2$ as being:

```
1  ProcEx2(b_NAT) =
     let
3       MemoryNATVar(n,b_NAT) = ...
        MemoryNAT(b_NAT) = ...
5       Memory(b_NAT) = MemoryNAT(b_NAT)
     within ( a -> b -> SKIP |~| a -> c -> SKIP )
```

Which is indeed the same behavior as of Beg's translated version $ACTION_FINAL$:

```
ACTION_FINAL(y,x) = a -> b -> SKIP |~| a -> c -> SKIP
```

And therefore, both processes are equivalent.

```
1  ArshadProEx2 = ACTION_FINAL(valueNAT(apply(b_NAT1,sv_x)),valueNAT(apply(b_NAT1,sv_y)))
   CTOCProcEx2 = ProcEx2(b_NAT1)
3  assert ArshadProEx2 [FD= CTOCProcEx2 -- Passed
   assert CTOCProcEx2 [FD= ArshadProEx2 -- Passed
```

Example 4 – Multiple assignments in an Action

A slightly longer process is defined as $ProcEx4$, which contains a larger number of assignments, when compared with Example 2.

process $ProcEx4 \mathrel{\widehat=}$
begin
  state $Ex4State \mathrel{\widehat=} [x, y : NatValue]$
  $InitProcEx4 \mathrel{\widehat=} (x := 1)\,;\ (y := 1)$
  $ActionA \mathrel{\widehat=} (x := 0)\,;\ (a \rightarrow \text{Skip})$
  $ActionB \mathrel{\widehat=} (y := x + 1 \bmod 5)\,;\ (b \rightarrow \text{Skip})$

$$ActionC \mathrel{\widehat=} \left(\begin{array}{l} (x := y + 1 \bmod 5)\,; \\ (y := y + x \bmod 5)\,; \\ (x := 4)\,; \\ (y := 5)\,; \\ (c \rightarrow \text{Skip}) \end{array}\right)$$

  $\bullet\ (ActionA\,;\ (ActionB\,;\ ActionC))$
end

We can see that some assignments does not mention any state variable, whereas others like $x := y + 1$ do require the values of such variables. Therefore, sometimes a *mget* event occurs prior to the *mset* event, for retrieving the current value of the variables mentioned in the assignment.

```
ProcEx4(b_NAT) =
  let
     MemoryNATVar(n,b_NAT) = ...
     MemoryNAT(b_NAT) = ...
     Memory(b_NAT) = MemoryNAT(b_NAT)
  within ( ( ( mset.sv_x.(NAT.0) ->
    a ->
    mget.sv_x?v_sv_x:(typeNAT(sv_x)) ->
    mset.sv_y.(NAT.((valueNAT(v_sv_x) + 1)%5)) ->
    b ->
    mget.sv_y?v_sv_y:(typeNAT(sv_y)) ->
    mset.sv_x.(NAT.(valueNAT(v_sv_y) + 1)) ->
    mget.sv_x?v_sv_x:(typeNAT(sv_x)) ->
    mget.sv_y?v_sv_y:(typeNAT(sv_y)) ->
    mset.sv_y.(NAT.(valueNAT(v_sv_y) + valueNAT(v_sv_x))) ->
    mset.sv_x.(NAT.4) ->
    mset.sv_y.(NAT.5) ->
    c -> SKIP;
      terminate -> SKIP )
      [| MEMI |] Memory(b_NAT))\MEMI )
```

We can observe in the translation from Beg that sometimes the behavior of the action does not clearly show when a value is updated. For instance, in $ActionA$, we first have an assignment, $x := 0$, then an event a followed by *Skip*. Therefore, we expect the variables update prior to the event, which does not occur as shown in the first two lines of the following code:

```
ACTION_A4(y,x) = a -> ACTION_B4(y,0 )
ACTION_B4(y,x) = b -> ACTION_C_0((x + 1)%5 ,x)
ACTION_C_0(y,x) = ACTION_C_1(y,(y - 1) )
ACTION_C_1(y,x) = ACTION_C_2((y + x)%5,x)
ACTION_C_2(y,x) = ACTION_C_3(y,4)
ACTION_C_3(y,x) = ACTION_C_4(5,x)
ACTION_C_4(y,x) = c -> SKIP
```

We see that first, an event a occurs and then the value of x is updated to 0. Similarly, the $ActionB$ behaves as described in $ActionA$. However, for $ActionC$, all assignments, and therefore, state transitions, occurs prior to the event c, which is the final one, $ACTION_C_4$, as described above.

Our approach using *Circus2CSP*, however, preserves the sequence of state variable updates. Finally, as the events a, b, and c, does not mention any state variable, the behavior of our translation is indeed equivalent to Beg's approach.

```
ArshadProEx4 = ACTION_A4(valueNAT(apply(b_NAT1,sv_x)),valueNAT(apply(b_NAT1,sv_y)))
CTOCProcEx4 = ProcEx4(b_NAT1)
assert ArshadProEx4 [FD= CTOCProcEx4 -- Passed
assert CTOCProcEx4 [FD= ArshadProEx4 -- Passed
```

Example 5 – Assignment Does Not Resolve Choice - Assignment in One Side

This example introduces an interesting discussion about the non-interference of state changes in

choice.

$$
\begin{array}{l}
\textbf{process}\ ProcEx5 \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ Ex5State \mathrel{\widehat=} [x, y : NatValue] \\
\quad InitProcEx5 \mathrel{\widehat=} (x := 1) \ ; \ (y := 1) \\
\quad ActionA \mathrel{\widehat=} (x := 0) \ ; \ (a \rightarrow \mathrm{Skip}) \\
\quad ActionB \mathrel{\widehat=} (y := x + 1 \bmod 5) \ ; \ (b \rightarrow \mathrm{Skip}) \\
\quad ActionC \mathrel{\widehat=} c \rightarrow \mathrm{Skip} \\
\quad \bullet\ ActionA \ ; \ (ActionB \Box ActionC) \\
\textbf{end}
\end{array}
$$

In the example above, as part of the main action, there is an external choice between $ActionB$ and $ActionC$, which then may be seen as:

$$ProcEx5.(\ldots ; \ (((y := x + 1 \bmod 5) \ ; \ \ b \rightarrow \mathrm{Skip}) \Box (c \rightarrow \mathrm{Skip})))$$

Such case is described by Oliveira [129, Definition B.7, p. 35], where he argues that since the process state is encapsulated, and therefore, should be invisible to the global environment, any state change should also not be visible, and therefore, an assignment should not interfere in the choice of events.

As part of our research, we were able to prove[1] that property using *Circus* refinement laws combined with the omega transformations. As we know that assignments are not available in CSP_M, we translate them into a sequence of $mget$ and $mset$ events, which are hidden to the external environment. Using the refinement laws we were able to shift the $mget$ and $mset$ events to occur after the event choice. Therefore, based on the Proof G.2.2, the above external choice becomes:

$$ProcEx5.(\ldots \ ; \ ((b \rightarrow mset.sv_y.((v_sv_x + 1) \bmod 5) \rightarrow \mathrm{Skip}) \Box (c \rightarrow \mathrm{Skip})))$$

Thus, as expected, the CSP_M code of the $ProcEx5$ translates the *Circus* process using the combination of the refinement laws and omega transformations, resulting in the same behavior as described above, where the choice must be resolved between the events b and c and depending on them, the value of y may be updated.

```
ProcEx5(b_NAT) =
  let
     MemoryNATVar(n,b_NAT) = ...
     MemoryNAT(b_NAT) = ...
     Memory(b_NAT) = MemoryNAT(b_NAT)
  within ( ( (
    mset.sv_x.(NAT.0) -> a -> -- ActionA
    mget.sv_x?v_sv_x:(typeNAT(sv_x)) ->
    mget.sv_y?v_sv_y:(typeNAT(sv_y)) ->
    (
     b -> mset.sv_y.(NAT.((valueNAT(v_sv_x) + 1)%5)) -> SKIP -- ActionB
     []
     c -> SKIP -- ActionC
     );
    terminate -> SKIP )
    [| MEMI |] Memory(b_NAT))\MEMI )
```

Finally, as expected, both translations of $ProcEx5$ are equivalent to each other.

```
ACTION_FINAL5(y,x) = a-> BOX(y,0 )
BOX(y,x) = b-> SKIP [] c-> SKIP

ArshadProEx5 = ACTION_FINAL5(valueNAT(apply(b_NAT1,sv_x)),
```

[1]The entire proof can be found in the Appendix G.2

```
                                valueNAT(apply(b_NAT1,sv_y)))
CTOCProcEx5 = ProcEx5(b_NAT1)

assert ArshadProEx5 [FD= CTOCProcEx5 -- Passed
assert CTOCProcEx5 [FD= ArshadProEx5 -- Passed
```

Example 6 – Assignment Does Not Resolve Choice - Asssignment on Both Sides

This is a second case of the non-interference of assignments on external choice. Here, assignments are used in both sides of the external choice, and according to the Proof G.2.1, the property holds, which shows that the final values of both x and y depends on the occurence of the events, whichever was chosed.

$$
\begin{array}{l}
\mathbf{process}\ ProcEx6 \mathrel{\widehat=} \\
\mathbf{begin} \\
\quad \mathbf{state}\ Ex6State \mathrel{\widehat=} [x, y : NatValue] \\
\quad InitProcEx6 \mathrel{\widehat=} (x := 1) \, ; \, (y := 1) \\
\quad ActionA \mathrel{\widehat=} (x := 0) \, ; \, (a \rightarrow \mathrm{Skip}) \\
\quad ActionB \mathrel{\widehat=} (y := x + 1) \, ; \, (b \rightarrow \mathrm{Skip}) \\
\quad ActionC \mathrel{\widehat=} c \rightarrow \mathrm{Skip} \\
\quad \bullet\ (ActionA \mathbin{\Box} ActionB) \, ; \, ActionC \\
\mathbf{end}
\end{array}
$$

Therefore, using *Circus2CSP*, we obtain the following CSP_M code.

```
ProcEx6(b_NAT) =
  let
     MemoryNATVar(n,b_NAT) = ...
     MemoryNAT(b_NAT) = ...
     Memory(b_NAT) = MemoryNAT(b_NAT)
  within ( ( (
    mget.sv_x?v_sv_x:(typeNAT(sv_x)) ->
    mget.sv_y?v_sv_y:(typeNAT(sv_y)) ->
    (
      (
        a -> mset.sv_x.(NAT.0) -> SKIP -- ActionA
        []
        b -> mset.sv_y.(NAT.((valueNAT(v_sv_x) + 1)%5)) -> SKIP -- ActionB
      );
      c -> SKIP -- ActionC
    );
    terminate -> SKIP )
      [| MEMI |] Memory(b_NAT))\MEMI )
```

The resulting CSP_M code translated by Beg as described in his thesis is defined below. However, we see no pattern for state changes after the occurrence of a or b, leading it straight to the following code, rather than to an intermediate action before the execution of the event c.

```
  ACTION_FINAL6(y,x) = (a -> b -> SKIP) [] (a -> c -> SKIP)
```

However, we believe there is a mistake as he argues that sequential composition distributes over external choice,

$$
\begin{array}{l}
(ActionA \mathbin{\Box} ActionB) \, ; \, ActionC \\
\quad \sqsubseteq (ActionA \, ; \, ActionC) \mathbin{\Box} (ActionB \, ; \, ActionC) \\
\quad \sqsubseteq \left(\begin{array}{l} (\ (x := 0 \, ; \, a \rightarrow \mathrm{Skip}) \, ; \, c \rightarrow \mathrm{Skip}\ ) \\ \Box \\ (\ (y := x + 1 \, ; \, b \rightarrow \mathrm{Skip}) \, ; \, c \rightarrow \mathrm{Skip}\ ) \end{array} \right)
\end{array}
$$

and therefore, it should result, instead, in the definition below. Moreover, we also used FDR4 to assert the equivalence between the two models.

```
ACTION_A_0(y,x) = ACTION_A_1(y,0)
ACTION_A_1(y,x) = a -> ACTION_C(y,x)
ACTION_B_0(y,x) = ACTION_B_1(y,x+1)
ACTION_B_1(y,x) = b -> ACTION_C(y,x)
ACTION_C(y,x) = c -> SKIP
ACTION_FINAL6(y,x) = ACTION_A_0(y,x) [] ACTION_B_0(y,x) -- Possible correct CSPm

ArshadProEx6 = ACTION_FINAL6(valueNAT(apply(b_NAT1,sv_x)),
                             valueNAT(apply(b_NAT1,sv_y)))

CTOCProcEx6 = ProcEx6(b_NAT1)

assert ArshadProEx6 [FD= CTOCProcEx6 -- Passed
assert CTOCProcEx6 [FD= ArshadProEx6 -- Passed
```

In the next section, we introduce examples used in order to evaluate the various improvements to the translation strategies, as previously presented, and we point out when such improvements were made necessary.

7.2 Evaluating *Circus2CSP* with Case-Studies

While experimenting with the translated CSP_M specifications using FDR4, our main goal was to obtain a model to which FDR4 can check, avoiding state-space explosion or excessive checking times. We report here some experiments done with the HD machine and the ring-buffer case study from D24.1. Our experiments make comparisons between the following translation schemes:

byHand The original hand translation of the HD model performed for the ABZ case-study.

D241 An automated translation based on that described in Compass deliverable D24.1

typedD241 A decomposition of `D241`, where each type has its own *BINDINGS* set and only refers to the state variables of that type.

CTOC The final automated translation that provides *Circus* models with one memory process per variable.

All these results were obtained on an *Intel Core i7* 2.8GHz CPU with 16GB of RAM. We also have tested our tool with several existing examples, like the air controller [14], among several other examples of *Circus* specifications also used for testing the *JCircus* tool. We were able to produce a translation for the chronometer models from Oliveira's PhD thesis [129]. As an experiment, we were able to assert, in FDR, that both *AChrono*, and *Chrono*, are equivalent: *AChrono* is refined by *Chrono* and vice-versa. Its refinement is also proved manually by Oliveira [129, p.34-41]. The translation and refinement checks in FDR of the chronometer example are detailed in Appendix H. In the next section, we describe our experiments with our tool, regarding the *Memory* model refinements using the Haemodialysis case study [75].

7.3 The Haemodialysis Case Study

Haemodialysis is a therapy used in order to purify the blood of patients diagnosed with kidneys failure. Such therapy is usually performed using a machine which removes waste such as creatinine and urea, as well as an excess of water, which is usually excreted when the kidneys are fully functional. The

dialysis machine uses a semipermeable membrane, in between the patient's blood flow and the dialysate solution, through which the blood's impurities are transferred to the dialysate solution, which in turn, is then discarded at the end of the therapy.

Such a device reveals the need for a deep understanding of the field of nephrology, as well as detailed documentation of the hemodialysis machine requirements, which involves continuous monitoring of dozens of safety-critical parameters, such as blood flow rates, pH, temperature and air detection. Moreover, the system requirements are complicated as they rely on the values of those parameters. Now, we briefly introduce *Circus* using some pieces of the Haemodialysis machine specification [75].

7.3.1 The Haemodialysis Machine in *Circus*

A *Circus* specification is composed of a series of paragraphs with can be either Z paragraphs, CSP processes, or mixed CSP with commands, defining *Circus* actions. *Circus* also uses Z schemas in order to model the "state" variables of a process, which might be read or updated using *Circus* actions.

One of the state components is the Z schema *HDGenComp*, composed of the state variables that we identified whilst reading the system requirements. For instance, we include the *hdActivity* and *infSalVol* parameters from **R-1**.

$$HDGenComp \mathrel{\widehat=} [hdActivity : \mathbb{P}\ ACTIVITY \land infSalVol : \mathbb{N} \land \ldots]$$

We now define the *Circus* process *HDMachine*, which captures all the state variables in "***state*** *HDState*", being composed of a conjunction of the *HDGenComp* schema, along with *RinsingParameters* and all the other schemas, such as *DFParameters*, *UFParameters*, *PressureParameters*, and *HeparinParameters*, modelling the variables detailed in [Mashkoor2016, Tables 3–6].

$$\begin{array}{l}\textbf{process}\ HDMachine \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\ HDState == \left(\begin{array}{l} HDGenComp \land RinsingParameters\\ \land DFParameters \land UFParameters\\ \land PressureParameters \land HeparinParameters\end{array}\right)\end{array}$$

Then, the behavior of the system is captured using *Circus* actions. Firstly, we describe the execution flow of the system from the therapy preparation to the conclusion, as illustrated in Figure 7.1. According to the requirements, the system starts with the preparation phase, followed by the initiation phase, and an ending phase, which are defined as the *MainTherapy* action, as a sequence of three *Circus* actions: *TherapyPreparation*, *TherapyInitiation* and *TherapyEnding*.

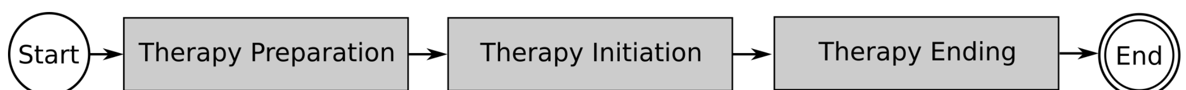


Figure 7.1: Execution flow of the Haemodialysis machine

$$MainTherapy \mathrel{\widehat=} TherapyPreparation\ ;\ TherapyInitiation\ ;\ TherapyEnding$$

As an example, for the therapy preparation phase, we define a *Circus* action that starts with a signal *preparationPhase*, followed by a sequence of activities according to [Mashkoor2016,§3.2]. A key idea here is each phase signals that it has started, on a channel, so that requirements and activities that are phase-dependent can ascertain when they should be active. We model the steps that compose

the therapy preparation phase, with a *Circus* process that behaves accordingly. The remaining two top-level actions, *TherapyInitiation* and *TherapyEnding* are modelled similarly and can be found in [75].

$$
\begin{array}{l}
\mathit{TherapyPreparation} \mathrel{\widehat=} \\
\quad \mathit{preparationPhase} \rightarrow \mathit{AutomatedSelfTest}; \\
\quad \mathit{ConnectingTheConcentrate} \; ; \; \mathit{SetRinsingParameters}; \\
\quad \mathit{InsertingRinsingTestingTubSystem} \; ; \; \mathit{PrepHeparinPump}; \\
\quad \mathit{SetTreatParameters} \; ; \; \mathit{RinsingDialyzer}
\end{array}
$$

7.3.2 System Requirements

We capture the system requirements by writing *Circus* actions that describe the behaviour of each of them, and then, we run such actions in parallel with the *MainTherapy* action, synchronising on common events, in order to allow the system to enforce raising alarms or stopping the therapy whenever the system execution detects unexpected readings from the various sensors. As an example of how we model the requirements, we describe **R-1**:

> *During the application of arterial bolus, the system monitors the volume of saline infusion and if the volume exceeds 400ml, the system should stop the blood flow and raise an alarm signal.*

Therefore, in *Circus*, we use conditionals where if the above restriction is satisfied, the system will perform *StopBloodFlow* and *RaiseAlarm*. If the precondition is not satisfied, it waits for a defined period (parameter *CheckInterval*) and checks again. This illustrates the general approach here for many of these monitoring requirements.

$$
R1 \mathrel{\widehat=} \left(
\begin{array}{l}
\mathbf{if} \\
\left(\begin{array}{l} (\; \mathit{hdActivity} = \{\mathit{appArtBolus}\} \land \mathit{infSalVol} > 400 \;) \\ \qquad \longrightarrow (\mathit{StopBloodFlow} \mid\mid\mid \mathit{RaiseAlarm}) \end{array} \right) \\
[\!] \left(\begin{array}{l} \neg \; (\; \mathit{hdActivity} = \{\mathit{appArtBolus}\} \land \mathit{infSalVol} > 400 \;) \\ \qquad \longrightarrow (\; \mathit{Wait}(\mathit{CheckInterval}) \rightarrow R1) \end{array} \right) \\
\mathbf{fi}
\end{array}
\right)
$$

Finally, the main action to be executed in the *Circus* process *HDMachine* is defined as *HDMain*, where the system variables are initialised with *HDGentCompInit*, followed by the main therapy interleaved with the requirements, which we restrict for illustration purposes to the Arterial Bolus section.

$$
\begin{array}{l}
\quad \mathit{HDMain} \mathrel{\widehat=} \mathit{HDGenCompInit} \; ; \; (\mathit{MainTherapy} \mid\mid\mid R1) \\
\quad\; \bullet \; \mathit{HDMain} \\
\mathbf{end}
\end{array}
$$

In this section, we briefly introduced the Haemodialysis case study along with some components of its *Circus* specification, as presented in [75]. In the next section, we discuss various approaches for model checking *Circus*.

7.3.3 Translating it into CSP_M

In order to model check the *Circus* specification of the Haemodialysis machine, our first attempt was to manually translate the *Circus* specification into CSP_M, which we will refer to as `byHand` in this

thesis. After the ABZ'16, we started the development of a tool based on the translation strategy developed for the EU COMPASS project and described in deliverable D24.1[132] (herein `D241`).

However, we identified that `D241` does not support specifications that use mixed types: it would not support our Haemodialysis machine, as we use several free types, as well as subsets of the natural numbers. Moreover, we noticed that such an approach is not suitable for *Circus* processes with a large number of state variables: our HD machine model has more than 20 state variables.

In order to overcome the limitations of `D241`, we use the *Circus* refinement laws in order to produce an improved translation scheme, which we will refer to as `CTOC` in this thesis. In this section, we provide an overview of `byHand`, and then, we present the translation using *Circus2CSP*.

A manual approach

We present here a few pieces from the CSP_M model of the Haemodialysis machine as presented in ABZ'16, using `byHand`. First, we translate the Z types into equivalent constructs in CSP_M. For instance, natural numbers are translated into a set of meaningful values, as we do not want FDR to explore all possible values, but only those being used. Therefore, we use $NatValue = \{0..X\}$ for the definition of natural numbers. Moreover, free types are translated into datatypes in CSP_M. For instance, the type $ACTIVITY$ is used for exploring the various tasks performed by the HD machine, such as the application of arterial bolus, $appArtBolus$ and the reinfusion process, $reinfProcess$.

```
datatype ACTIVITY = reinfProcess | appArtBolus | ...
```

Then, Z schemas are translated by defining name types in CSP_M, which consists of producing tuples with the values for each variable from the schema. For example, the schema $HDGenComp$ is translated into a tuple where the first component of the tuple is the range of values for the $hdActivity$, of type $\mathbb{P}\,ACTIVITY$, translated into CSP_M as `Set.(ACTIVITY)`. For model-checking purposes, we redefine the limits for the $infSalVol$ ranging from 0 up to 1, as we have to limit the range of values using `byHand`.

```
1 nametype HDGenComp = (Set.(ACTIVITY),{0..1},...)
```

In *Circus*, when we want to manipulate the values of a component of a state component, we can freely access it, even using assignments, as they are used within the context of the *Circus* process. However, the main issue while translating *Circus* into CSP_M is the fact that manipulating state variables is not possible in CSP_M as we do in *Circus* and assignments are not present in the syntax of CSP.

Therefore, the approach in order to overcome such problems, works as follows: if we want, for example, to obtain the value of $hdActivity$, we define an auxiliary function $get_hdActivity$ illustrated below, using as input a tuple of type $HDGenComp$ and which returns the value of the corresponding component. Similarly, an assignment such as $hdActivity := val$ is made possible by calling the function $set_hdActivity(val, (hda, inf, ...))$, where val is the value to be assigned replacing hda and $(hda, inf, ...)$ is the tuple of type $HDGenComp$, as illustrated below.

```
get_hdActivity((hda,inf,...)) = hda
set_hdActivity(val,(hda,inf,...)) = (val,inf,...)
```

The overall shape of the *HDMachine* process translated using `byHand` is presented as follows: it has a tuple as a parameter, where each of the state components is structured within the equivalent notation for the Z schemas. We represent the state of the process by creating new processes, where every component of the *Circus* state is defined as the input of that process in CSP_M and the access to those values is made through CSP_M channels, allowing the main action of the process to access and update the values of the state components.

For instance, the *HDGenComp* is defined as the first member of the tuple, *HDGC*. Then, in order to read and write each state variable, we create internal processes, like as *HDGenCompSt* for the *HDGenComp*: each variable is read using the channels prefixed with *get*, and is updated through channels prefixed with *set*. For instance, *getHdActivity* provides the current value of *hdActivity*, and *setHdActivity* receives the its value and recurses updating it in *HDGC*.

```
HDMachine((HDGC, RP, DFP, UFP, PP, HP)) =
  let
  HDGenCompSt(HDGC) =
    getHdActivity!get_hdActivity(HDGC) -> HDGenCompSt(HDGC)
    [] setHdActivity!vHd -> HDGenCompSt(set_hdActivity(vHd, HDGC))
    [] ...
```

The requirement **R-1** is defined in CSP_M similarly to the *Circus* model. However, as there are no direct access to the state variables, the values of *hdActivity* and *infSalVol* is obtained through *getHdActivity* and *getinfSalVol* respectively, prior to the `if-then-else` construct. Moreover, the behavior of **R-1** is preserved during the translation.

```
R1 = getHdActivity?hdActivity -> getinfSalVol?infSalVol ->
    if hdActivity == {appArtBolus} and infSalVol > 0
    then (StopBloodFlow ||| RaiseAlarm)
    else Wait(CheckInterval) ; R1
```

The main therapy of the Haemodialysis machine, modelled *MainTherapy* is translated as a sequence of *TherapyPreparation*, followed by *therapyInitiation* and concluded with *TherapyEnding*.

```
MainTherapy = TherapyPreparation ; TherapyInitiation ; TherapyEnding
```

The structure of the CSP_M process is defined as the execution of *HDGenCompInit*, initialising the *HDGenComp* variables, followed by the *MainTherapy* in parallel with *R*1. In order to be able to read and update the values of the state variables, the main behavior is put in parallel with *HDGenCompSt*, synchronising on channel set *HDComm*, composed of the *get*−prefixed and *set*−prefixed channels.

```
  within ((HDGenCompInit ; (R1 ||| MainTherapy))
              [|HDComm|] HDGenCompSt(HDGC)) \ HDComm
```

This concludes the translation of the haemodialysis machine using `byHand`. The resulting CSP_M file has twice the number of lines compared to its *Circus* specification written in LaTeX. Such strategy

applied to this case study justifies the need for an automated translation as the *Circus* specification is already large, and we had to manually create auxiliary functions, as well as channels, used for the *gets* and *sets* between the actions and the state variables in the CSP_M process. In the next subsubsection, we present the translated model using *Circus2CSP*.

Automatic Translation using *Circus2CSP*

We present now a short description of the CSP_M code of the haemodialysis machine from the *Circus* model [75] generated using the strategy CTOC. We inserted comments in the code below, identifying a few actions which were expanded into the main action of the process. For instance, the action *TherapyPreparation* can be found in line 12, as well as the scope of the requirement R-1 is defined between the lines 34 and 45.

```
HDMachine(b_NAT,...) =
  let
    MemoryNATVar(n,b_NAT) = mget.n.apply(b_NAT,n) -> MemoryNATVar(n,b_NAT)
      [] mset.n?nv:typeNAT(n) -> MemoryNATVar(n,over(b_NAT,n,nv))
      [] terminate -> SKIP
    MemoryNAT(b_NAT) =
      ([|{| terminate |}|] n : dom(b_NAT) @ MemoryNATVar(n,b_NAT))
    ...
    Memory(b_NAT,...) = (MemoryNAT(b_NAT) [|{| terminate |}|] Memory...)
    MemoryMerge(b_NAT,...) = ...
    within ( ( ( (
    preparationPhase -> autSelfTest -> -- TherapyPreparation action
    mset.sv_signalLamp.(BOO.TRUE) ->
    connectingConcentrate?x -> mset.sv_bicarbonateAcetate.(BOO.x) ->
    ( ( ( (
      ( atrialTubing -> SKIP ||| ventricularTubing -> SKIP );
      salineBagLevel?ifs ->
      mset.sv_infSalineVol.(NAT.ifs) -> SKIP );
      insertHeparinSyringe ->
      heparinLineIsVented -> SKIP );
      connectDialyzer ->
      fillArterialDrip -> stopBP -> SKIP );
      therapyInitiation -> connectingToPatient ->
      mset.sv_signalLamp.(BOO.FALSE) ->
      connPatientArterially ->
      setbloodFlowInEBC?bf -> mset.sv_bloodFlowInEBC.(NAT.bf) ->
      connPatientVenously ->
      mset.sv_signalLamp.(BOO.TRUE) ->
      ...
      mset.sv_signalLamp.(BOO.FALSE) ->
      therapyEnding -> -- TherapyEnding action
      mset.sv_signalLamp.(BOO.FALSE) -> SKIP )
      |||
      ( let muR1 = -- Begin of R-1
              mget.sv_hdActivity?v_sv_hdActivity:(typePBO(sv_hdActivity)) ->
              mget.sv_infSalineVol?v_sv_infSalineVol:(typeNAT(sv_infSalineVol)) ->
              (((valuePBO(v_sv_hdActivity) == {TRUE})
                and (valueNAT(v_sv_infSalineVol) > 0)) &
                  ( stopBloodFlow -> SKIP
                    ||| mset.sv_alarm.(BOO.TRUE) -> produceAlarmSound -> SKIP )
              [] not (((valuePBO(v_sv_hdActivity) == {TRUE})
                      and (valueNAT(v_sv_infSalineVol) > 0))) &
                  tick -> muR1)
        within muR1 ) -- End of R-1
    ); terminate -> SKIP )
      [| MEMI |] Memory(b_PBO,b_BOO,b_NAT))\MEMI )
```

In the next section, we describe and discuss our experiments on the haemodialysis machine using the above-presented approaches, reinforcing the relevance of our translation strategy implemented in our tool, *Circus2CSP*.

7.3.4 HD Machine Experiments

In this section we discuss the use of our translation tool with the example of the Haemodialysis case study, detailing the results of using the several approaches for model checking *Circus* specifications

translated into CSP_M and then using FDR. Our first model was presented in the ABZ'16 [75], where a manual translation (`byHand`) of the haemodialysis machine *Circus* specification was made, and then we were able to verify it in FDR. Such an approach, however, is not practical as the manual translation is time-consuming and may introduce errors.

As a result of the translation `byHand` for the haemodialysis machine, the number of lines in the CSP_M file doubled (2264 lines) compared to the LaTeX *Circus* specification (1024 lines). Moreover, 124 new auxiliary functions (`get_` and `set_`) were created, used for setting and getting the values of the state variables in the name type tuples. Besides, 113 channels were used for synchronising values between the main action and the state process were created.

Then, as we implemented the tool based on Oliveira's approach [132], we identified that the haemodialysis machine model in CSP_M leads to a type error in FDR as it does not support polymorphic function definitions, which is the case of the `type` function. We, therefore, were unable to perform any check of the HD machine in FDR using `D241`. The results of experiments on the intermediate steps in translation improvement between `D241` and `CTOC` will be presented in the sequel.

The solution to the problem of `D241` came with a series of refinements in the translation scheme. Firstly, we defined `typedD241`, where the bindings used in the specification were split into subsets related to the types used of the state variables, as presented in Section 4.1.1.

We solved the issue of `typedD241` with a new approach, where the refinement laws were used in order to refine the *Circus* process, moving the bindings, offered initially as local variables, to parameters of the process. Therefore, FDR would not have to check all possible bindings, but only one, provided by the user. In our tool, we developed a mechanism able to generate a single binding as an example for model checking. Such a requirement is feasible as we expect the *Circus* specification, and therefore the translated CSP_M specification, to initialise all the state variables.

For the first time using *Circus2CSP*, we were able to obtain results from FDR regarding our *HDMachine*. However, due to a large number of state variables, the state exploration in FDR was taking several minutes to conclude the checks. We concluded that the *Memory* model used was generating a large number of available *mset* and *mget* synchronisations that were never being used. Therefore, we designed `CTOC`, a new approach with a *Memory* model firstly distributed by its type, and then, for each type-related memory process, we have smaller processes, where each state variable is manipulated (through *mget* and *mset*) independently. Details on such an improvement were presented in Section 4.1.2.

Our reference *Circus* model was that of the haemodialysis machine running in parallel with a model of one of its requirements (**R-1** [7, Section 4.2, p11]). The requirement model is effectively a monitor that observes the machine model, checking that it is satisfied, and deadlocking if it observes a violation. We then check the proposition that the HD model is correct w.r.t **R-1** by showing that the combination is deadlock free. In addition to comparing various translation schemes, we also explored the effect of changing the size of our "natural number" type:

$NatValue == 0 \mathrel{..} N$

We explored the `byHand` and `CTOC` translation schemes with nine ranges of *NatValue* size, with N up to a maximum of 90, as shown in in Table 7.1[2]. The only case where we could compare the two approaches was our first case, with $N = 1$: it resulted in 9,409 states visited using `byHand`, in contrast with 811 states visited using `CTOC`, resulting in a reduction of 91% regarding states explored. Moreover, the execution time with the model generated using `CTOC` was equally reduced by 91% compared to the model using `byHand`.

The "Plys" column indicates how deep the breadth-first search algorithm used by FDR went while checking. The number of plys is larger for the `byHand` model and is independent of the value of N. Interestingly, after waiting more than 2 hours, we were unable to obtain results from the model generated with `byHand` when we increased the N to 2. However, the model generated with `CTOC`, when tested using $n = 90$, was executed in 35 seconds, which is still quicker than `byHand` with $N = 1$. We also note that the amount of memory used was constant, at 240MB approx.

Table 7.1: Time for asserting deadlock freedom of the HD Machine in FDR4

Approach	NatValue range	Result	States Visited	Transitions Visited	Plys Visited	Exec. Time
	0..1	Passed	811	1,800	39	0.375s
	0..2	Passed	1,761	3,786	39	0.407s
	0..4	Passed	4,645	9,834	39	0.420s
	0..6	Passed	8,841	18,650	39	0.508s
	0..8	Passed	14,349	30,234	39	0.602s
	0..10	Passed	21,169	44,586	39	0.937s
	0..20	Passed	74,949	157,866	39	2.352s
CTOC	0..30	Passed	161,529	340,346	39	3.465s
	0..90	Passed	1,369,809	1,369,809	39	35.097s
	0..1	Passed	9,409	301,617	47	40.826s
byHand	0..2	incomplete	?	?	?	$>$ 2 hours

In addition to experiments that varied N above, we also explored how the *number* of variables, rather than the size of their datatypes, influenced the checking time. Using a hypothetical example not related to the haemodialysis case study having 12 state variables, checks using `D241` were performed in 35 minutes, compared to 76 ms using `CTOC`. We observed segmentation faults using `D241` with more than 12 variables. However, checks using `CTOC` in an example with 42 state variables and $NatValue = 0 \mathrel{..} 30$, were performed in 870 ms.

What is clear is that with the `CTOC` translation scheme, namely one memory-process per state-variable, we can now handle *Circus* models of considerable complexity, based on the results obtained from model-checking the haemodialysis machine specification in FDR. Using the `CTOC` translation results in CSP_M with approximately 75% fewer lines compared to the `byHand` translation approach.

7.4 Ring-Buffer Experiments

Another interesting example was to take the *Circus* specification of the bounded reactive ring buffer, RB, from D24.1 [132, Appendix D.2, p. 163], based on the model presented in [35]. We compared the `CTOC` translation of this using *Circus2CSP* (RB_{CTOC}), with the by-hand translation in D24.1 [132,

[2]All these results were obtained on an *Intel Core i7* 2.8GHz CPU with 16GB of RAM, and no compression techniques were used in FDR for this experiment.

Appendix D.4, p166] (RB_{D241}). We also compared it with the model of the ring buffer, RB_{KW}, based on [172, Chapter 22], produced using the approach of Kangfeng and Woodcock [174] for translating *Circus* into `CSP||B`, which is similar to the one from [132, p. 116] but makes use of Z schemas. We firstly perform the usual tests like deadlock freedom and termination checks for all three specifications, as illustrated in Table 7.2.

Table 7.2: *RingBuffer* checks: deadlock and livelock freedom, and determinism.

Test	Model	Result	States Visited	Transitions	Plys	Exec.Time
deadlock free	RB_{D241}	Passed	8,297,025	16,805,249	44	26.657s
	RB_{CTOC}	Passed	1,628	3,109	38	0.145s
	RB_{KW}	Failed	248	713	8	0.128s
deterministic	RB_{D241}	Passed	9,869,889	19,852,673	69	54.863s
	RB_{CTOC}	Passed	2,012	3,853	63	0.159s
	RB_{KW}	Failed	27	84	3	0.156s

We can see a clear difference between the states visited between the three approaches, notably those between RB_{byH} and RB_{CTOC} where the number of states and transitions visited was reduced considerably, as well as the amount of time spent by FDR4 to check the assertions. However, the tests performed with the CSP_M specification of RB_{KW} failed the checks for deadlock freedom and fdeterminism. Such results are expected since the RB_{KW} model is translated into `CSP||B`, and therefore, some properties of the specification are defined in B, and can be evaluated properly by ProB, but not by FDR4, which sees only the CSP_M specification files.

We also experimented to check the failures-divergences refinement ($P \sqsubseteq_{FD} Q$) between the three approaches, each pair in both directions, as described in Table 7.3. Since we know that the specification RB_{CTOC} is a translation from the same *Circus* model of the handmade translation of RB_{byH}, we expect that RB_{byH} and RB_{CTOC} are equivalent to each other, $RB_{byH} \sqsubseteq_{FD} RB_{CTOC}$ and $RB_{CTOC} \sqsubseteq_{FD} RB_{byH}$, which is true, as seen below in row 1 and 3. However, the model RB_{KW} is refined by both RB_{CTOC} and RB_{byH} (rows 2 and 4), but the refinement in the reverse direction does not hold (rows 5 and 6), *i.e.*, RB_{KW} is not a refinement of neither RB_{CTOC} nor RB_{byH}, as it is a more abstract model since part of the whole RB_{KW} model is defined in B.

Table 7.3: Refinement checks between three models of the Ring Buffer example

	Assertion				States Visited	Transitions Visited	Plys Visited	Exec. time
1	RB_{byH}	$\sqsubseteq_{FD}$	RB_{CTOC}	✓	1,628	3,109	38	58.019s
2	RB_{KW}	$\sqsubseteq_{FD}$	RB_{CTOC}	✓	1,733	3,365	43	0.226s
3	RB_{CTOC}	$\sqsubseteq_{FD}$	RB_{byH}	✓	8,297,025	16,805,249	44	42.543s
4	RB_{KW}	$\sqsubseteq_{FD}$	RB_{byH}	✓	8,809,345	18,087,809	49	43.939s
5	RB_{byH}	$\sqsubseteq_{FD}$	RB_{KW}	×	27	98	3	57.680s
6	RB_{CTOC}	$\sqsubseteq_{FD}$	RB_{KW}	×	27	91	3	0.172s

Interestingly, if we compare the states and transitions visited, as well as the execution time from Table 7.2 with Table 7.3, given a refinement $A \sqsubseteq_{FD} B$, the states and transitions visited are almost the same as of checking B for deadlock freedom. However, we noticed that the execution time of any refinement check that involves RB_{byH} are much longer than those between RB_{CTOC} and RB_{KW}.

Unfortunately, the structure defined for this translation strategy is not fully supported by ProB [101], which was used to test RB_{KW} [174]. ProB is another model-checker, which like FDR4, also allows the user to animate specifications. It was originally developed for the B language, but it has been extended and now it supports other formal languages such as CSP, Z, Event-B [1], as well as combined languages such as `CSP||B`. We observed that the use of `subtype`, in our models, is not fully supported by the ProB tool, causing some commands like "model-check" to result in errors. However, we were able to animate our translated specification using ProB, and to execute the same assertion check, as in FDR4: our experiments with ProB demonstrated similar results to those obtained with FDR4.

7.5 Compression Experiments

An important aspect when using FDR is the availability of compression techniques [142] in order to reduce the number of states, improving the time required for refinement checking. A compression transforms a labelled-transition system (LTS) into a corresponding one, which is expected to be smaller and more efficient whilst using it for checks in FDR. The current version of FDR, 4.2, applies compressions in parallel compositions by default, which is the main structure we use in our memory model. We explored a few compression tecniques, such as *sbisim*, which determines the maximal strong bisimulation [25], and *wbisim*, which computes the maximal weak bisimulation. Depending on the compression used, the number of states visited, or plys visited[3], were indeed reduced, as illustrated in Table 7.4.

Table 7.4: Experimenting CSP_M compression techniques with the HD Machine

	sbisim+diamond		no compression		sbisim		wbisim	
Values Range	States Visited	Exec Time (seconds)	States Visited	Exec Time (seconds)	States Visited	Exec Time (seconds)	States Visited	Exec Time (seconds)
0..10	77	0.499	21,169	0.458	302	0.479	87	0.56
0..20	77	0.986	74,949	0.819	302	0.925	87	1.106
0..40	77	2.901	280,909	2.263	302	2.582	87	3.647
0..80	77	11.093	1,086,429	8.043	302	10.013	87	13.343
0..120	77	25.096	2,416,749	18.805	302	21.793	87	35.839
0..160	77	47.635	4,271,869	35.148	302	42.267	87	70.011
0..240	77	114.845	9,556,509	84.803	302	100.112	87	175.846
0..360	77	327.815	21,419,469	235.236	302	269.414	killed	286.079
0..480	77	668.437	38,005,629	467.602	302	523.825	killed	525.889

Although the states visited were considerably reduced using the compression techniques mentioned above, the time consumption is almost the same for small range of values for the *NatValue* type. However, as we increase that range, we see that the states visited does not change for those cases when compression techniques were used, but we noticed that the time spent is much longer than the model checked with no use of compression techniques. Finally, we noticed that FDR was unable to perform the checks using *wbisim* for the cases where the range of *NatValue* were larger than 0..240, being killed by the operating system.

[3]Number of plys that were visited in the breadth-first search.

7.6 Final considerations for testing *Circus2CSP*

There is one thing we need to have in mind when performing model checking of *Circus* specifications: we need to avoid state explosion. In FDR, the translated specifications of a *Circus* model into CSP can cause state explosion due to the infinite number of possible states of the system. We then have to restrict, in the CSP specifications, the types to a small subset of the original types specified in *Circus*. Since we can deal with a large number of processes in parallel and their communication channels, some of the assertions would require too much computing resources and could lead to a system crash due to lack of system memory.

In order to overcome such problem, another possible contribution for this work could be the development of a module for the translation tool that would be able to produce a subset of the possible range of values that FDR would use for testing. For example, given a type $TIME = \{0..59\}$ and a variable $second : TIME$, if the variable $second$ is used once in the entire specification, lets us consider the expression $second > 15$ being evaluated. We do not need to evaluate every single value of the range $\{0..59\}$, but only two, for instance, 15 and 16. Therefore, a suitable subset would be $second : \{15, 16\}$. Such a tool would analyze FDR3 much simpler since the state space would be significantly smaller and the assertions would be more easily evaluated.

In summary, we described in this chapter our experiments performed throughout the timeline of the development process of *Circus2CSP*. We used the case studies presented here as testing set in order to identify any weakness in the translation approach, as we presented in the previous chapters. Moreover, we used such examples as a way of evaluating the tool capabilities, the scalability of the approach, and to be able to evaluate the efficiency of FDR while checking the translated specifications using the various translation approaches implemented until now.

Chapter 8

Validating the Translation

Besides the development of *Circus2CSP*, we would like to reason about the correctness of our approach, including the improvements provided during our research. One of our first tasks is to formally verify the refinement of the memory model from Section 4.

8.1 Verifying the Refinements on the Memory Model

As discussed earlier in this thesis, the Memory model from the model as presented for `D241` was refined into the distributed model used in our tool `CTOC`. For such, we use the *Circus* refinement laws and we were able to manually prove the equivalence (bidirectional refinement) between the original *Memory* process used in `D241` (renamed in this section to $Memory_{D241}$) and the partitioned memory process used in `CTOC` ($Memory_{CTOC}$), as presented below.

$$
\begin{array}{l}
Memory_{D241} \mathrel{\widehat=} \mathbf{vres}\; b : BINDING \bullet \\
\quad \left(\begin{array}{l}
(\Box\, n : \mathrm{dom}\, b \bullet mget.n!b(n) \rightarrow Memory_{D241}(b)) \\
\Box \left(\begin{array}{l}
\Box\, n : \mathrm{dom}\, b \bullet mset.n?nv : (nv \in \delta(n)) \rightarrow \\
\qquad Memory_{D241}(b \oplus \{n \mapsto nv\})
\end{array}\right) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
Memory_{CTOC} \mathrel{\widehat=} \mathbf{var}\; b : BINDING \bullet \\
\quad (\llbracket \{\!|\, terminate \,|\!\} \rrbracket\; n : \mathrm{dom}\, b \bullet Memory_{Var}(n, n \lhd b))
\end{array}
$$

$$
\begin{array}{l}
Memory_{Var} \mathrel{\widehat=} \mathbf{var}\; n : NAME;\ b : BINDING \bullet \\
\quad \left(\begin{array}{l}
mget.n!b(n) \rightarrow Memory_{Var}(n, b) \\
\Box\, mset.n?nv : (nv \in \delta(n)) \rightarrow Memory_{Var}(n, (b \oplus \{n \mapsto nv\})) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)
\end{array}
$$

In order to show that both memory models are equivalent, we proved that:

$$Memory_{D241} \sqsubseteq Memory_{CTOC}$$

and

$$Memory_{CTOC} \sqsubseteq Memory_{D241}$$

8.1.1 Memory Model Refinement Proof

We present here the proof steps for the refinement of the memory model. Such steps look at the simple case where there are only two state variables, n_1 and n_2, in order to keep things concise. However, the proofs steps are essentially the same for any number of state variables.

$$Memory_{CTOC}(b)$$

$= \hfill$ [$Memory_{CTOC}$ definition]

$$\llbracket \{\!|\, terminate \,|\!\} \rrbracket\ n : \mathrm{dom}\ b \bullet Memory_{Var}(n, n \lhd b)$$

$= \hfill$ [Iterated parallelism definition]

$$Memory_{Var}(n_1, n_1 \lhd b)\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, n_2 \lhd b)$$

$= \hfill$ [$Memory_{Var}$ definition]

$$\left(\begin{array}{l}
mget.n_1!b(n_1) \rightarrow Memory_{Var}(n_1, n_1 \lhd b) \\
\Box\, mset.n_1?nv : (nv \in \delta(n_1)) \rightarrow Memory_{Var}(n_1, n_1 \lhd (b \oplus \{n_1 \mapsto nv\})) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)$$
$$\llbracket \{\!|\, terminate \,|\!\} \rrbracket$$
$$\left(\begin{array}{l}
mget.n_2!b(n_2) \rightarrow Memory_{Var}(n_2, n_2 \lhd b) \\
\Box\, mset.n_2?nv : (nv \in \delta(n_2)) \rightarrow Memory_{Var}(n_2, n_2 \lhd (b \oplus \{n_2 \mapsto nv\})) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)$$

$= \hfill$ [[82, L7 - p.51]]

$$\left(\begin{array}{l}
mget.n_1!b(n_1) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_1, n_1 \lhd b) \\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, n_2 \lhd b) \end{array}\right) \\
\Box\, mget.n_2!b(n_2) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_2, n_2 \lhd b) \\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_1, n_1 \lhd b) \end{array}\right) \\
\Box\, mset.n_1?nv : (nv \in \delta(n_1)) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_1, n_1 \lhd (b \oplus \{n_1 \mapsto nv\})) \\ \quad \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, (n_2 \lhd b))) \end{array}\right) \\
\Box\, mset.n_2?nv : (nv \in \delta(n_2)) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_2, n_2 \lhd (b \oplus \{n_2 \mapsto nv\})) \\ \quad \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_1, (n_1 \lhd b))) \end{array}\right) \\
\Box\, terminate \rightarrow (\mathrm{Skip}\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ \mathrm{Skip})
\end{array}\right)$$

$= \hfill$ [Parallelism Composition Unit - [129, Law C.90]]

$$\left(\begin{array}{l}
mget.n_1!b(n_1) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_1, n_1 \lhd b) \\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, n_2 \lhd b) \end{array}\right) \\
\Box\, mget.n_2!b(n_2) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_2, n_2 \lhd b) \\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_1, n_1 \lhd b) \end{array}\right) \\
\Box\, mset.n_1?nv : (nv \in \delta(n_1)) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_1, n_1 \lhd (b \oplus \{n_1 \mapsto nv\})) \\ \quad \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, (n_2 \lhd b))) \end{array}\right) \\
\Box\, mset.n_2?nv : (nv \in \delta(n_2)) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_2, n_2 \lhd (b \oplus \{n_2 \mapsto nv\})) \\ \quad \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_1, (n_1 \lhd b))) \end{array}\right) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)$$

$= \hfill$ [$(n_2 \lhd (b \oplus \{n_1 \mapsto nv\})) = n_2 \lhd b$ and $(n_1 \lhd (b \oplus \{n_2 \mapsto nv\})) = n_1 \lhd b$]

$$\left(\begin{array}{l}
mget.n_1!b(n_1) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_1, n_1 \lhd b) \\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, n_2 \lhd b) \end{array}\right) \\
\Box\, mget.n_2!b(n_2) \rightarrow \left(\begin{array}{l} Memory_{Var}(n_2, n_2 \lhd b) \\ \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_1, n_1 \lhd b) \end{array}\right) \\
\Box \left(\begin{array}{l} mset.n_1?nv : (nv \in \delta(n_1)) \rightarrow \\ \quad Memory_{Var}(n_1, (n_1 \lhd (b \oplus \{n_1 \mapsto nv\}))) \\ \quad \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_2, (n_2 \lhd (b \oplus \{n_1 \mapsto nv\}))) \end{array}\right) \\
\Box \left(\begin{array}{l} mset.n_2?nv : (nv \in \delta(n_2)) \rightarrow \\ \quad Memory_{Var}(n_2, (n_2 \lhd (b \oplus \{n_2 \mapsto nv\}))) \\ \quad \llbracket \{\!|\, terminate \,|\!\} \rrbracket\ Memory_{Var}(n_1, (n_1 \lhd (b \oplus \{n_2 \mapsto nv\}))) \end{array}\right) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)$$

$= \hfill$ [$Memory_{CTOC}$ definition]

$$\left(\begin{array}{l}
mget.n_1!b(n_1) \rightarrow Memory_{CTOC}(b) \\
\Box\, mget.n_2!b(n_2) \rightarrow Memory_{CTOC}(b) \\
\Box\, mset.n_1?nv : (nv \in \delta(n_1)) \rightarrow Memory_{CTOC}(b \oplus \{n_1 \mapsto nv\}) \\
\Box\, mset.n_2?nv : (nv \in \delta(n_2)) \rightarrow Memory_{CTOC}(b \oplus \{n_2 \mapsto nv\}) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)$$

$= \hfill$ [Iterated External Choice definition]

$$\left(\begin{array}{l}
(\Box\, n : \mathrm{dom}\ b \bullet mget.n!b(n) \rightarrow Memory_{CTOC}(b)) \\
\Box\, (\Box\, n : \mathrm{dom}\ b \bullet mset.n?nv : (nv \in \delta(n)) \rightarrow Memory_{CTOC}(b \oplus \{n \mapsto nv\})) \\
\Box\, terminate \rightarrow \mathrm{Skip}
\end{array}\right)$$

$$= \quad \text{[Same (guarded) recursive form as } Memory_{D241}\text{, so unique fixed-point law applies]}$$

$$\left(\begin{array}{l}(\Box\, n : \operatorname{dom} b \bullet mget.n!b(n) \rightarrow Memory_{D241}(b)) \\ \Box\, (\Box\, n : \operatorname{dom} b \bullet mset.n?nv : (nv \in \delta(n)) \rightarrow Memory_{D241}(b \oplus \{n \mapsto nv\})) \\ \Box\, terminate \rightarrow \text{Skip}\end{array}\right)$$

$$= \quad [Memory_{D241} \text{ definition}]$$

$$Memory_{D241}(b)$$

8.1.2 Memory Model Refinement using FDR

We used FDR in order to verify every refinement step presented above. Our intention was to split the refinement steps into intermediate CSP_M processes, and use the failures-divergence refinement showing the equivalence between each two steps, *i.e.*, asserting the refinement in both directions. In total, we have six refinement steps, defined as `LHS`, `S1`, `S2`, `S3`, `S4` and `RHS`, where `LHS` is the CSP_M description of $Memory_{D241}$ and `RHS` is the equivalent model to $Memory_{CTOC}$.

We illustrate the CSP_M code used for the refinement steps below, where the final two assertions asserts if `LHS` is refined by `RHS` and also if `RHS` is refined by `LHS`. All the assertions below passed through the tests from FDR, proving that the refinement is true, and therefore, the models are equivalent. For the sake of simplicity, we used only two variables v_1 and v_2, but such an approach can be extended to more variables.

```
  LHS = MemoryNAT(b_nat)
--
assert LHS [FD= S1
assert S1 [FD= LHS
--
  S1 = ( ( mget.v_1.apply(b_nat,v_1) -> MemoryNATVar(v_1,dres(b_nat,{v_1}))
          [] mset.v_1?nv:typeNAT(v_1) -> MemoryNATVar(v_1,dres(over(b_nat,v_1,nv),{v_1}))
          [] terminate -> SKIP)
       [|{|terminate|}|]
       ( ( mget.v_2.apply(b_nat,v_2) -> MemoryNATVar(v_2,dres(b_nat,{v_2}))
         [] mset.v_2?nv:typeNAT(v_2) -> MemoryNATVar(v_2,dres(over(b_nat,v_2,nv),{v_2})))
         [] terminate -> SKIP))
--
assert S2 [FD= S3
assert S3 [FD= S2
--
  S3 =
       mget.v_1.apply(b_nat,v_1) -> (MemoryNATVar(v_1,dres(b_nat,{v_1}))
                                     [|{|terminate|}|]
                                     MemoryNATVar(v_2,dres(b_nat,{v_2})))
    [] mget.v_2.apply(b_nat,v_2) -> (MemoryNATVar(v_2,dres(b_nat,{v_2}))
                                     [|{|terminate|}|]
                                     MemoryNATVar(v_1,dres(b_nat,{v_1})))
    [] mset.v_1?nv:typeNAT(v_1) -> (MemoryNATVar(v_1,dres(over(b_nat,v_1,nv),{v_1}))
                                     [|{|terminate|}|]
                                     MemoryNATVar(v_2,dres(over(b_nat,v_1,nv),{v_2})))
    [] mset.v_2?nv:typeNAT(v_2) -> (MemoryNATVar(v_2,dres(over(b_nat,v_2,nv),{v_2}))
                                     [|{|terminate|}|]
                                     MemoryNATVar(v_1,dres(over(b_nat,v_2,nv),{v_1})))
    [] terminate -> (SKIP [|{|terminate|}|] SKIP)
--
assert S2 [FD= S3
assert S3 [FD= S2
--
  S3 =
       mget.v_1.apply(b_nat,v_1) -> MemoryNAT(b_nat)
    [] mget.v_2.apply(b_nat,v_2) -> MemoryNAT(b_nat)
    [] mset.v_1?nv:typeNAT(v_1) -> MemoryNAT(over(b_nat,v_1,nv))
    [] mset.v_2?nv:typeNAT(v_2) -> MemoryNAT(over(b_nat,v_2,nv))
    [] terminate -> SKIP
--
assert S3 [FD= S4
assert S4 [FD= S3
--
  S4 =
    ([] n:dom(b_nat) @ mget.n.apply(b_nat,n) -> MemoryNAT(b_nat))
    [] ([] n:dom(b_nat) @ mset.n?nv:typeNAT(n) -> MemoryNAT(over(b_nat,n,nv)))
    [] terminate -> SKIP
--
assert S4 [FD= RHS
assert RHS [FD= S4
```

```
52  --
    RHS =
54    ([] n:dom(b_nat) @ mget.n.apply(b_nat,n) -> Memory(b_nat))
      [] ([] n:dom(b_nat) @ mset.n?nv:typeNAT(n) -> Memory(over(b_nat,n,nv)))
56    [] terminate -> SKIP
    --
58  -- Showing the equivalence of LHS with RHS
    assert LHS [FD= RHS
60  assert RHS [FD= LHS
```

Our research aims to go beyond the implementation of the translation tool. We present in the next section some plans for future work regarding experiments on the verification of our translation approach.

8.2 Plans for the Verified Translation

We intend to link our tool with other tools as we did with FDR. For a future work, we would like to create a path to theorem provers like Isabelle/HOL. A key question is how to ensure that the translations we have developed are correct with respect to the semantics of both *Circus* and CSP. We would like to verify the correctness of our Haskell implementation of such a strategy.

In the long term, our goal towards this would be to, hopefully, link *Circus2CSP* with Isabelle/UTP [58, 59, 66] and be able to prove that the translation provided by our tool is equivalent to a manually refined *Circus* specification using Isabelle/UTP. Our expectation is that such an effort would improve the confidence of our translation so it can be used in industrial-scale programs.

We illustrate in Figure 8.1 the possible paths for showing that the translation approaches in both Haskell and Isabelle/UTP are equivalent. We use the function *toUTP* in order to transform the Haskell representation of *Circus* specifications into Isabelle/UTP. Moreover, we would have to design a function in Isabelle/HOL, seen in the figure below as *refUTP*, in order to be able to map the Haskell implementation of each refinement step into its corresponding representation in Isabelle/UTP.

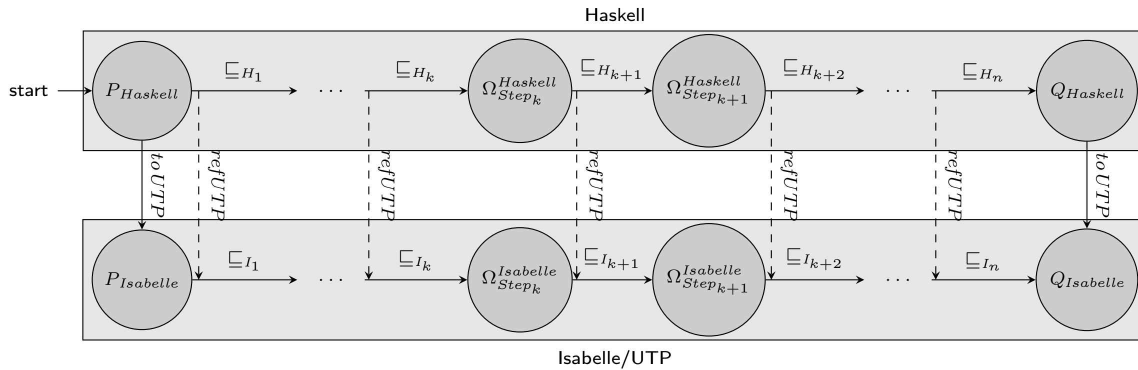


Figure 8.1: Plans for verifying the correctness of *Circus2CSP*

In order to verify the correctness of our implementation of the translation from *Circus* to CSP_M, as mentioned in Section 3.1, we want to prove that the refinement steps implemented in Haskell, herein Ω_H, from staterich *Circus* processes ($Circus_{SR}$) to stateless *Circus* processes ($Circus_{SL}$) are correct *w.r.t.* those refinement steps implemented in Isabelle/UTP, represented here as Ω_I. We hightlight that the notation used here for Ω_H and Ω_I represents not only the Ω functions as presented in Chapter 4, but also the relevant *Circus* refinement laws used during such a refinement process.

Unfortunately, the current version of Isabelle/UTP does not support all of the *Circus* operators used in our translation strategy. The stateless *Circus* models produced using *Circus2CSP* uses parametric recursion, hiding and complex event structures in the prefix constructs. We know that mechanising those constructs in Isabelle/UTP is tricky and would require a considerable effort for specifying them and proving its properties in Isabelle.

We attempted to work closely with the Isabelle/UTP development team at the University of York, but after some interactions, we observed that such work goes beyond the scope of our research for this PhD. In order to contribute to the development of Isabelle/UTP with the formalisation of those constructs, we would need to expand our knowledge on the theories as defined in Isabelle, as well as to obtain a better understanding on the notation and style of proofs adopted by the other members of the development team. Some proofs already made are not trivial, and the Isar language for proofs in Isabelle might sometimes become very difficult to understand. Such an effort would require some time for learning as well as much more interaction with the current team.

During our research, a lot has been done aiming at linking our implementation to Isabelle/HOL theorem prover. However, the attempt to verify our tool was left as future work, along with the integration of the refinement calculator in order to discharge the proof obligations for both *Circus* refinements, as well as for the strategy for refining Z schemas into *Circus* actions, presented in Section 3.5. Some of our efforts for attempting to link our tool with Isabelle are described in Appendix E.

From the Requirement 3.1 presented in Section 3.1, we want to show that every Ω refinement step within a refinement from state-rich to stateless *Circus* processes should be proved to be correct using Isabelle. However, until now, we were not able to prove such a requirement, and thereore, it was left as one of our future work plans.

We could have implemented the translation strategy directly in Isabelle/HOL and perform the verification approach within the same platform. However, our idea for verifying our implementation came after our first version of the tool written in Haskell, when we started the refinements of the memory model, described in Section 4, along with the inclusion of new translation rules. Therefore, we decided to use Haskabelle in order to import our code into Isabelle. Moreover, we also had a Z parser within Jaza, and the effort for parsing *Circus* from LaTeX would still take time. Finally, we would either way have to export CSP_M code from the refinements performed, which would probably require the same implementation effort.

Moreover, we also would like to use a theorem prover to experiment with the proofs made in order to solve interesting problems in *Circus*. For instance, we would like to use a theorem prover to formalise our proof on the relation between assignments and external choice. As stated by Oliveira [129], an assignment does not resolve an external choice. However, when implementing the translation to CSP_M, we noticed that because assignments are transformed into a sequence of events ($mget$ and $mset$), and from the fact that any state change is not visible externally, we can shift an assignment after the events that are to be selected in the choice. We were able to prove such a proposition, as

presented below, by hand, where the entire proof can be found in Appendix G.2.

$$\Omega(P_S.((x := 0\ ;\ \ c_1 \rightarrow \text{Skip}) \,\Box\, (x := 1\ ;\ \ c_2 \rightarrow \text{Skip})))$$
$$=$$
$$P_S.((c_1 \rightarrow x := 0) \,\Box\, (c_2 \rightarrow x := 1))$$

In summary, our tool may be seen as a laboratory for experimenting and further contributions, since it was designed in such a way that other modules may be introduced and therefore, the range of experiments it can support may grow in the future. We hope that being able to link our tool with Isabelle theorem prover will improve the confidence of the community for using *Circus* as a formal modelling language, and motivate its use in industry. Finally, our tool may also be extended in order to be able to support test-cases generation.

Chapter 9

Conclusions

The work presented in this document was motivated by the need for a tool capable of model-checking specifications designed using *Circus*. As presented in Section 2.2, there has been an effort from the community in order to design a systematic approach for model-checking *Circus*, which due to its combination of formalisms, has been a challenge until now. The choice of the subject and the language itself comes from personal experiences with *Circus* for the past twelve years since the beginning of our experience with research as an undergraduate student in Brazil. The real challenge for model-checking *Circus* came during our research on the formalisation of the Integrated Modular Avionics architecture, following the ARINC 653 standards [74], where a manual translation of that model into CSP_M was generated, and required months of work due to the complexity of such a large model.

This PhD project was initially defined as an attempt to produce verified $Occam - \pi$ code from *Circus*, aiming at simulating programs using microcontrollers, such as Arduino boards. However, as we started our research, we felt the need for a tool capable of model-checking *Circus* in large-scale, before the refinement into code. Moreover, we know from the literature that other research has been done in order to obtain Java code from *Circus*, but the need for model-checking *Circus*was still a gap yet to be filled. Therefore, we decided that, for the moment, the work presented here would be of a much more valuable contribution than the proposed one for obtaining $Occam - \pi$ through refinement.

In the next section we summarise our contributions to the current state of the art, and then we conclude this document with directions of future work.

9.1 Contributions

We developed *Circus2CSP* by extending Jaza [162], a Z animator written in Haskell, to cover the *Circus* abstract syntax. Consequently, we can parse *Circus* specifications written in LaTeX, which is the notation adopted as the standard for the *Circus* community. We then implemented the translation strategy introduced by Oliveira *et al.* [132, Section 5.3, p.79], and used the *Circus* refinement laws, to improve both the scope and effectiveness of the translation. Finally, our tool also includes an automatic *Circus* refinement calculator, where we implemented the laws listed in Appendix A of the Deliverable 24.1 [132, p.147].

Our tool *Circus2CSP* is based on the set of rules for refining state-rich *Circus* into stateless processes that can be mapped into CSP_M [132]. The reason we adopted the translation presented by

Oliveira *et al.*, is that, even though it is a manual translation, with no tool support involved, each translation step is justified by the *Circus* refinement laws, which have been formally proved to be correct. Currently, their approach covers a subset of *Circus*. However, as presented in Section 4.1.1, our investigation through experiments with the implementation of such rules demonstrated that such an initial and theoretical approach was restricted for only a subset of the possible *Circus* specifications: those dealing with only one same type for all variables within the state of those processes.

Our contribution here begun to become clear, when we had to implement not only a tool for the translation but also to refine that translation strategy in order to support a more realistic set of specifications: those using mixed types among their state variables.

We also experimented with the efficiency of FDR concerning the scale of the specifications. For such, we used the haemodialysis case study, a complex system which behaves depending on the values of dozen of state variables. Thus, we refined the memory model in order to optimise the task of reading and updating the state variables from the *Circus* processes.

The modifications for the memory model presented here are similar to what was presented by Mota *et al.* [127], where interleaving between processes, one for each state variable, was proposed. In fact, the memory model used in [132] was based on the one by Mota *et al.*, and was expanded with the inclusion of a *terminate* signal, and, rather than one process for each variable, it would offer all possible *mget* and *mset* for all state variables at the same time. However, after implementing such model, we identified that such approach tends to make the job of FDR harder and expensive, leading to state space explosion, when using several state variables on the same memory model. In our research, we also seek ways of optimising the CSP_M model in order to obtain a more efficient analysis. Therefore, even though the *terminate* event is used for synchronising the end of the *Memory* execution, the parallel composition of all possible *mgets* and *msets* leads to a possible exponential growth of the state space, which was observed in Section 4.1.2. However, such a problem does not occur while using interleaving. Moreover, Mota *et al.* also argues that the use of interleaving helps the compression algorithms built in FDR [141] to reduce the state space exploration while analysing such models.

Moreover, throughout our research, we observed that we could produce a more concrete model when moving the decision of using the bindings to the model designer, where now we have processes using bindings given as parameters, instead of provided non-deterministically. As discussed in the refinement step 4.8, Chapter 4, we rely on the fact that the state variables should be initialised before its main execution [4]. As an experiment, we developed a first mechanism that provides one single instance of bindings as part of the CSP_M specification. The approach of parametrised bindings resulted in a significant reduction in the time spent by FDR while checking the models produced by *Circus2CSP*, as illustrated in Fig. 3.5 and Table 7.2. The outcome is that we now have a mechanised translator from *Circus* to CSP_M that produces tractable models, and allows the use of FDR on more extensive case studies that have been possible up to now.

We used the haemodialysis machine and the ring buffer case studies as examples in order to test the capabilities of our tool while model-checking the automatically translated models in FDR. We aimed to contribute to reducing FDR's workload in order to model check larger systems. We learned

that a practical implementation/mechanisation of a theory might reveal difficulties that could not otherwise be discovered without extensive use of a tool prototype, especially when applying it to more extensive case studies.

As part of the translation strategy proposed by Oliveira *et al.* [132], the *Circus* refinement laws were used during the refinement into the stateless *Circus* processes. We implemented an automatic refinement calculator as part of *Circus2CSP*, as presented in Section 3.4, which handles a selected set of laws used according to [132, Appendix A, p. 147]. Moreover, we experimented with a strategy for refining Z schemas into "schema-free" *Circus* actions using Z Refinement Calculus [34].

With *Circus2CSP*, we can model-check specifications written in *Circus* using FDR. Our tool is currently integrated with the command-line interface of FDR4, and, as presented in Chapter 6, the user can run FDR from within *Circus2CSP*. Moreover, we provide a way of writing CSP_M code in the LaTeX source files where the *Circus* specification is written, using the environment `assert`. Finally, our tool can automatically generate assertion checks about the currently loaded specification and perform a combination of refinement checks, along with checks for deadlock freedom, as well as deterministic check and divergence-free checks. However, so far, our tool is only able to provide binary answers, staring if the assertion has passed or not. We plan for the future to parse the counter-examples provided by FDR into our tool.

9.2 Future Work

We have plans to extend the coverage of the tool, looking for ways of overcoming the restrictions presented here, in particular, for those presented in Section 2.5. We have a particular interest in specifying a translation strategy for Z schemas used as *Circus* actions within a process, as presented in Section 3.5. The best approach would be to use Z Refinement Calculus [34]. For now, our tool deals only with those schemas that in fact can be translated into assignments. We intend to explore the operators for Z schemas and the refinement laws that can be applied accordingly.

The binding generator presented in Section 5.2 is a prototype of a more ambitious plan for future work where we intend to evaluate the specification looking for a suitable set of values for the types. Our idea is to find an approach that could restrict the range of values as much as possible, so FDR would only need to explore just a few significant values. For example, in the specification of the Chronometer, we discussed the restriction to a range of values from 0 up to 60. For model-checking purposes, we can consider an even more restricted range, as we are only interested in the interactions when the seconds are either 0 or one step before the 59th tick, which is when the clock increments the minutes. Therefore, we could use a range from 0 up to 5, which may be enough for simulating the behaviour of the model.

Another example is the full range of values used in the haemodialysis case study [75]. Supposing a specific parameter is used within a range of 35ml and 45ml, we know that the system will respond to any monitored value below and above that range. Therefore, we would not need to have a check for any value from 0ml up to 100ml, but probably, only something like 34ml up to 46ml, since the

new range would simulate any value outside the predetermined parameter settings. Such an approach then reduces a combination of 101 values to only 12 values for FDR to explore within the state space.

Besides, we also plan to strengthen the link between the Haskell syntax of *Circus* and Isabelle/UTP [66]. As discussed previously in Chapter 8, the current release of Isabelle/UTP does not yet support all of the constructs used in the translated form. Our goal will be to prove that either refining the *Circus* specification in our tool and then importing into Isabelle/UTP, or, importing the specification into Isabelle/UTP first, and then using the refinement laws, would result in the same refined specification.

Our tool also has a *Circus* refinement "calculator" embedded in it, which implements the laws listed in Appendix A of the Deliverable 24.1 [132, p.147], which can easily be extended to the other refinement laws proved by Oliveira [129] in the near future. We also plan to integrate the refinement calculator in order to be able to discharge proof obligations with support from Isabelle/UTP. This also involves generating lemmas for the proof obligations generated during the refinement of Z schemas using Z Refinement Calculus.

Finally, in terms of improvement of our tool, comparing to other approaches [49], it would also be interesting to review the parser of Z and *Circus* in Jaza in order to rewrite their AST to be in conformance with the International Standards Organization (ISO) standards, ISO/IEC 13568:2002 [89], which describes the syntax, type system and semantics of Z formal notation. Such changes would, therefore, be propagated to the *Circus* syntax accordingly. Moreover, we would like to include the *libcspm* library[1] into *Circus2CSP* in order to be able to parse the relevant code included in our definition of the *assertion* LaTeX environment. Such an attempt would help a *Circus2CSP* user wishing to review any fault in the CSP_M specification translated from *Circus*. It would be interesting to provide a mechanism for back annotation where we would be able to identify which part of the *Circus* model is the source of the failure, incorrect behaviour, or even to help to identify the source of a deadlock or non-determinism.

[1] https://github.com/tomgr/libcspm

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 2010.

[3] Jean-Raymond Abrial, C. A. R. Hoare, and Pierre Chapron. *The B-Book.* Cambridge University Press, Cambridge, 1996.

[4] Incorporated (ARINC) Aeronautical Radio. ARINC 653: Avionics Application Standard Software Interface, nov 2006.

[5] V S Alagar and K Periyasamy. Vienna Development Method. *Specification of Software Systems*, 2011.

[6] Arduino. Arduino - Home, 2016.

[7] Mashkoor Atif. The hemodialysis machine case study. In Michael J Butler, Klaus-Dieter Schewe, Atif Mashkoor, and Miklós Biró, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9675 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2016.

[8] Ralph-Johan. Back, J. von (Joakim) Wright, F. B. Wright, J. Von/Schneider, and D. Gries. *Refinement calculus : a systematic introduction.* Springer, 1998.

[9] Ralph-Johan R. Back. On the correctness of refinement steps in program development, 1978.

[10] Frédéric Badeau and Arnaud Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. *... : Formal Specification and Development in Z and B*, pages 1–18, 2005.

[11] Thomas Ball, Byron Cook, Vladimir Levin, Sriram K Rajamani, and Technical Report. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*. 2004.

[12] Richard Banach, Huibiao Zhu, Wen Su, and Xiaofeng Wu. Continuous ASM, and a pacemaker sensing fragment. In John Derrick, John S Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael

Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7316 LNCS of *Lecture Notes in Computer Science*, pages 65–78. Springer, 2012.

[13] Victor Bandur, Peter Würtz Vinther Tran-Jørgensen, Miran Hasanagic, and Kenneth Guldbrandt Lausdahl. Code-generating VDM for Embedded Devices. *The 15th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering*, pages 1–15, 2017.

[14] S L M Barrocas and Marcel Vinicius Medeiros Oliveira. JCircus 2.0: an Extension of an Automatic Translator from Circus to Java. *Communicating Process Architectures*, 2012.

[15] Arshad Beg. *Translating from State-Rich to State-Poor Process Algebras.* PhD thesis, School of Computer Science and Statistics, Trinity College, University of Dublin, Ireland, 2016.

[16] Arshad Beg and Andrew Butterfield. Development of a prototype translator from Circus to CSPm. In *ICOSST 2015 - 2015 International Conference on Open Source Systems and Technologies, Proceedings*, pages 16–23, dec 2016.

[17] Patrick Behm, Paul Benoit, Alain Faivre, and Jean Marc Meynadier. Météor: A successful application of b in a large project. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999.

[18] Juan Bicarregui. On the verification of VDM specification and refinement with PVS. In *Proceedings 12th IEEE International Conference Automated Software Engineering*, pages 280–289. IEEE Computer Society, 1997.

[19] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings 1999 Design Automation Conference*, pages 317–320, New York, New York, USA, 1999. ACM Press.

[20] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceeding*, pages 193–207. 1999.

[21] Frédéric Boniol and Virginie Wiels. The Landing Gear System Case Study. In *Communications in Computer and Information Science*, 2014.

[22] Arne Borälv. The industrial success of verification tools based on Stålmarck's method, 1997.

[23] Scientific Corporation Boston. ALTRUA Pacemaker System Guide, 2008.

[24] Boston Scientific. PACEMAKER system specification. *Boston Scientific*, 2007.

[25] Alexandre Boulgakov, Thomas Gibson-Robinson, and A. W. Roscoe. Computing maximal weak and other bisimulations. *Formal Aspects of Computing*, 28(3):381–407, 2016.

[26] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 1993.

[27] Randal E Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *{ACM} Comput. Surv.*, 24(3):293–318, 1992.

[28] J.R. Burch, Edmund M. Clarke, Kenneth L McMillan, David L. Dill, and L.J. Hwang. Symbolic model checking: 1020 States and beyond. *Information and Computation*, 98(2):142–170, 1992.

[29] Michael Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, 3582:221–236, 2005.

[30] Andrew Butterfield and J. C. P. Woodcock. Semantic domains for handel-c. *Electronic Notes in Theoretical Computer Science*, 74(2):1–20, 2003.

[31] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015.

[32] David Carrington, D Duke, Roger Duke, Paul King, G Rose, and Graeme Smith. {Object-Z}: An object-oriented extension to {Z}. In Son T Vuong, editor, *Formal Description Techniques (FORTE '89), Vancouver*, pages 281–296. North-Holland, 1989.

[33] Gareth Carter, Rosemary Monahan Monahan, and Joseph M. Morris. Software refinement with perfect developer, 2005.

[34] Ana Cavalcanti and J. C. P. Woodcock. ZRC - A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267–289, mar 1998.

[35] Ana L. C. Cavalcanti, Augusto C.A. A Sampaio, and J. C.P. P. Woodcock. A Refinement Strategy for Circus. In *Formal Aspects of Computing*, volume 15, pages 146–181, nov 2003.

[36] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[37] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. State/Event-Based Software Model Checking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2999:128–147, 2004.

[38] Taolue Chen, Marco Diciolla, Marta Kwiatkowska, and Alexandru Mereacre. Quantitative verification of implantable cardiac pacemakers over hybrid heart models. *Information and Computation*, 236:87–101, 2014.

[39] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of synchronization skeletons for branching time temporal logic. In *Logics of Programs Workshop, New York, May 1981*, volume 131, pages 52–71, Berlin/Heidelberg, 1982. Springer-Verlag.

[40] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent system using temporal logic specifications. *POPL '83:Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, 1983.

[41] Edmund M. Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking, 1987.

[42] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics*, pages 176–194. 2001.

[43] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[44] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, sep 1994.

[45] Clearsy. Atelier B. http://www.atelierb.eu/, aug 2011.

[46] Madiel Conserva Filho and Marcel Vinicius Medeiros Oliveira. Implementing tactics of refinement in CRefine. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7504 LNCS of *Lecture Notes in Computer Science*, pages 342–351. Springer, 2012.

[47] Luis Diogo Couto. *On the Extensibility of Formal Methods Tools*. PhD thesis, Aarhus University, 2015.

[48] David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. In *Tools Exhibition Notes at Formal Methods Europe*. 2003.

[49] CZT Partners. Community {Z} tools. oct 2006.

[50] Deliverable Number D. Comprehensive Modelling for Advanced Systems of Systems Compositional Analysis and Design of CML Models Contributors : Editors :. (September):1–22, 2013.

[51] John Derrick and Eerke A. Boiten. Combining CSP and Object-Z. In *Refinement in Z and Object-Z*, pages 431–455. Springer London, London, 2014.

[52] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[53] Jin Song Dong, Ping Hao, Sheng Chao Qin, Jun Sun, and Wang Yi. - Timed Patterns: TCOZ to Timed Automata. In *International Conference on Formal Engineering Methods*, pages 483—-498. Springer, 2004.

[54] Jin Song Dong, Jun Sun, and Yang Liu. Build your own model checker in one month. In *Proceedings - International Conference on Software Engineering*, pages 1481–1483, 2013.

[55] Jérôme Falampin, Hung Le-Dang, Michael Leuschel, Mikael Mokrani, and Daniel Plagge. Improving Railway Data Validation with ProB. In *Industrial Deployment of System Engineering Methods*, pages 27–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[56] Adalberto C Farias, Alexandre Mota, and Augusto Sampaio. Efficient CSP{\itshape Z} Data Abstraction.

[57] Abderrahmane Feliachi, Marie-claude Gaudel, and Makarius Wenzel. Isabelle / Circus. pages 1–108, 2016.

[58] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying theories in isabelle/HOL. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6445 LNCS, pages 188–206. Springer, Berlin, Heidelberg, nov 2010.

[59] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle / Circus : a Process Specification and Verification Environment. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7152 LNCS, pages 243–260, 2012.

[60] Clemens Fischer. Combining Object-Z and CSP. In Adam Wolisz, Ina Schieferdecker, and Axel Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch, Berlin, 19.-20. Juni 1997*, volume 315 of *GMD-Studien*, pages 119–128. GMD-Forschungszentrum Informationstechnik GmbH, 1997.

[61] Clemens Fischer and Heike Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In K. Araki, A. Galloway, and K Taguchi, editors, *Ifm'99*, pages 315–334. Springer London, London, 1999.

[62] John S. Fitzgerald and Peter Gorm Larsen. *Modelling systems: practical tools and techniques in software development.* Cambridge University Press, 2009.

[63] John S. Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated designs for object-oriented systems.* Springer-Verlag, London, 2005.

[64] John S. Fitzgerald, Simon Tjell, Peter Gorm Larsen, and Marcel Verhoef. Validation Support for Distributed Real-Time Embedded Systems in VDM++. *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pages 331–340, 2007.

[65] Radio Technical Commission for Aeronautics. {DO-178C}, Software Considerations in Airborne Systems and Equipment Certification, 2011.

[66] Simon Fowler, Frank Zeyda, and J. C. P. Woodcock. Isabelle/UTP: {A} Mechanised Theory Engineering Framework. In *Unifying Theories of Programming - 5th International Symposium, {UTP} 2014, Singapore, May 13, 2014, Revised Selected Papers*, pages 21–41, 2014.

[67] Angela F. Freitas and Ana Cavalcanti. Automatic Translation from Circus to Java. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods: 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[68] Leonardo Freitas. *Model checking Circus.* PhD thesis, Department of Computer Science, The University of York, UK, 2005.

[69] Leonardo Freitas, J. C. P. Woodcock, and Ana Cavalcanti. State-rich model checking. *Innovations in Systems and Software Engineering*, 2(1):49–64, 2006.

[70] A.J. Galloway and W.J. Stoddart. An operational semantics for ZCCS. In *Proceedings First IEEE International Conference on Formal Engineering Methods*, number January 1998, pages 272–282. IEEE Comput. Sco, 1997.

[71] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 18(2):149–167, 2016.

[72] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew William Roscoe. FDR3 — A Modern Model Checker for CSP. *Tools and Algorithms for the Construction and Analysis of Systems*, 8413:187–201, 2014.

[73] Thomas Gibson-robinson and A. W. Roscoe. FDR into The Cloud. 1, 2014.

[74] Artur Oliveira Gomes. *Formal Specification of the ARINC 653 Architecture Using Circus.* PhD thesis, University of York, 2012.

[75] Artur Oliveira Gomes and Andrew Butterfield. Modelling the haemodialysis machine with Circus. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9675(201857):409–424, 2016.

[76] Artur Oliveira Gomes and Andrew Butterfield. Circus2CSP - A Translator from Circus to CSPm, 2018.

[77] Artur Oliveira Gomes and Marcel Vinicius Medeiros Oliveira. Formal development of a cardiac pacemaker: From specification to code. In Jim Davies, Leila Silva, and Adenilso da Silva Simão, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6527 LNCS of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2011.

[78] Artur Oliveira Gomes and Marcel Vinicius Medeiros Vinícius Medeiros Oliveira. Formal Specification of a Cardiac Pacing System. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 692–707. Springer, 2009.

[79] Florian Haftmann. From Higher-Order Logic to Haskell : There and Back Again. *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 155–158, 2010.

[80] Gerald H Hilderink, A. Bakkers, and Jan Broenink. A distributed real-time Java system based on CSP. In *Proceedings - 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000*, pages 400–407. IEEE Computer Society, 2000.

[81] Gerald H Hilderink, Jan Broenink, and André Bakkers. Communicating Java Threads. *Parallel Programming and Java, Proceedings of WoTUG 20*, 50:48–76, 1997.

[82] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[83] C.A.R. Hoare and J. He. *Unifying Theories of Programming.* Prentice-Hall, 1998.

[84] Gerard J Holzmann. The Model Checker Spin. *Ieee Transactions on Software Engineering*, 23(5):279–295, 1997.

[85] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed).* Addison-Wesley, 2003.

[86] David Hopkins and A. W. Roscoe. SVA , a tool for analysing shared-variable programs. *Electronic Notes in Theoretical Computer Science*, pages 1–5, 2007.

[87] MacColl Ian and David Carrington. Specifying Interactive Systems in Object-Z and CSP. In Araki Keijiro, Andy Galloway, and Taguchi Kenji, editors, *IFM'99: Proceedings of the 1st International Conference on Integrated Formal Methods, York, 28-29 June 1999*, pages 335–352, London, 1999. Springer London.

[88] Yoshinao Isobe and Markus Roggenbach. A Generic Theorem Prover of CSP Refinement. *Society*, pages 108–123, 2005.

[89] ISO/IEC. ISO/IEC 13568:2002 Information Technology âĂŞ Z formal specification notation âĂŞ Syntax, type system and semantics. Technical report, 2002.

[90] Ethan K. Jackson, Tihamér Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6981 LNCS of *Lecture Notes in Computer Science*, pages 653–667. Springer, oct 2011.

[91] Christian L. Jacobsen and Matthew C. Jadud. Towards concrete concurrency. *ACM SIGCSE Bulletin*, 37(1):431, feb 2005.

[92] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[93] Xiaoping Jia. Approach to animating Z specifications. In *Proceedings - IEEE Computer Society's International Computer Software & Applications Conference*, 1995.

[94] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. Modeling and verification of a dual chamber implantable pacemaker. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7214 LNCS, pages 188–203, 2012.

[95] Georgios Kanakis, Peter Gorm Larsen, Peter W.V.\ Tran-J\orgensen, Peter Würtz Vinther Tran-Jørgensen, and Peter W.V.\ Tran-J\orgensen. Code Generation of VDM++ Concurrency. *Proceedings of the 13th Overture Workshop*, (June):60–74, 2015.

[96] G Kassel and Graeme Smith. Model checking {Object-Z} classes: some experiments with {FDR}. *Eighth Asia-Pacific Software Engineering Conference*, pages 445–452, 2001.

[97] Tim G Kimber. Object-Z to Perfect Developer. Technical Report September, 2007.

[98] Vladimir Klebanov, Philipp Rümmer, Steffen Schlager, and Peter H Schmitt. {V}erification of {JCSP} {P}rograms. In Jan F Broenink, Herman Roebbers, Johan P E Sunter, Peter H Welch, and David C Wood, editors, *{C}ommunicating {P}rocess {A}rchitectures 2005*, pages 203–218, sep 2005.

[99] Sebastian Krings, Jens Bendisposto, and Michael Leuschel. From failure to proof: The PROB disprover for B and Event-B. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9276:199–214, 2015.

[100] Lukas Ladenberger, Dominik Hansen, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. Validation of the ABZ landing gear system using ProB. *International Journal on Software Tools for Technology Transfer*, 19(2):187–203, 2017.

[101] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. *FME 2003 Formal Methods*, 2805:855–874, 2003.

[102] Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.

[103] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale B models with ProB. *Formal Aspects of Computing*, 23(6):683–709, nov 2011.

[104] Gavin Lowe. Specification of communicating processes: Temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3):277–294, 2008.

[105] Formal Systems (Europe) Ltd. ProBE Manual., oct 2010.

[106] Lemma 1 Ltd. ProofPower. http://www.lemma-one.com/ProofPower/, aug 2009.

[107] Hugo Daniel Macedo, Peter Gorm Larsen, and John S. Fitzgerald. Incremental development of a distributed real-time model of a cardiac pacing system using VDM. In Jorge Cuéllar, T. S E Maibaum, and Kaisa Sere, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5014 LNCS of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.

[108] Brendan Mahony and Jin Song Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. *Proceedings of the 20th International Conference on Software Engineering*, pages 95–104, 1998.

[109] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. *ZB 2005: Formal Specification and Development in Z and B*, 3455:315–352, 2005.

[110] Kenneth L McMillan. Symbolic Model Checking. *Computer Aided Verification*, 1102:419–422, 1993.

[111] Dominique Méry, Bernhard Schätz, and Alan Wassyng. The Pacemaker Challenge: Developing Certifiable Medical Devices.

[112] Dominique Méry and Neeraj Kumar Singh. Functional Behavior of a Cardiac Pacing System. *International Journal of Discrete Event Control Systems*, 1(2):129–149, 2011.

[113] Dominique Méry and Neeraj Kumar Singh. Closed-loop modeling of cardiac pacemaker and heart. In Jens Weber and Isabelle Perseil, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7789 LNCS of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2013.

[114] Dominique Méry and Neeraj Kumar Singh. Formal specification of medical systems by proof-based refinement. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(1):15, 2013.

[115] Dominique Méry, Neeraj Kumar Singh, and Others. Technical Report on Formal Development of Two-electrode Cardiac Pacing System. *MOSEL (LORIA)*, 2010.

[116] Tim Miller, Leonardo Freitas, Petra Malik, and Mark Utting. CZT support for Z extensions. *Integrated Formal Methods*, pages 227–245, 2005.

[117] Robin Milner. A Calculus of Communicating Systems, 1980.

[118] J S Moores. *The design and implementation of OCCAM/CSP support for a range of languages and platforms.* PhD thesis, University of Kent, UK, 2002.

[119] Carroll Morgan. *Programming from specifications.* Prentice Hall, 1990.

[120] Carroll Morgan. The Refinement Calculus, and Literate Development. In *Formal Program Development*, pages 161–182, 1993.

[121] Carroll Morgan and Carroll. *Programming from specifications (2nd. Ed.)*, volume 16 of *Prentice Hall International series in computer science*. Prentice Hall, 1994.

[122] Alexandre Mota, Adalberto C Farias, André Didier, and Jim C. P. Woodcock. Rapid prototyping of a semantically well founded Circus model checker. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8702 LNCS of *Lecture Notes in Computer Science*, pages 235–249. Springer, 2014.

[123] Alexandre Mota, Adalberto C Farias, J. C. P. Woodcock, and Peter Gorm Larsen. Model checking CML: tool development and industrial applications. *Formal Aspects of Computing*, 27(5-6):975–1001, nov 2015.

[124] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z, 1998.

[125] Paul Mukherjee. System Refinement in VDM-SL. *Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'96)*, 00:483, 1996.

[126] Chris Nevison. Teaching distributed and parallel computing with Java and CSP. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 484–491. IEEE Comput. Soc, 2001.

[127] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects of Computing*, 26(3):441–490, 2014.

[128] Diego Oliveira and Marcel Vinicius Medeiros Oliveira. Joker: An Animation Framework for Formal Specications. In *SBMF'11 - Short Papers*, pages 43–48, ICMC/USP, sep 2011.

[129] Marcel Vinicius Medeiros Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, University of York, UK, 2005.

[130] Marcel Vinicius Medeiros Oliveira, Ana Cavalcanti, and J. C. P. Woodcock. Unifying theories in ProofPower-Z. In *Formal Aspects of Computing*, volume 25, pages 133–158, jan 2013.

[131] Marcel Vinicius Medeiros Oliveira, Alessandro Cavalcante Gurgel, and C. G. Castro. CRefine: Support for the Circus Refinement Calculus. In Antonio Cerone and Stefan Gruner, editors, *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 281–290. IEEE, 2008.

[132] Marcel Vinicius Medeiros Oliveira, Augusto Sampaio, Pedro Antonino, Rodrigo Ramos, Ana Cavalcanti, and J. C. P. Woodcock. Compositional Analysis and Design of CML Models. Technical Report D24.1, COMPASS Deliverable, 2013.

[133] Marcel Vinicius Medeiros Oliveira, Augusto Sampaio, and Madiel Conserva Filho. Model-Checking Circus State-Rich Specifications. *Integrated Formal Methods 2014*, 8739 LNCS:39–54, 2014.

[134] Marcel Vinicius Medeiros Oliveira, J. C. P. Woodcock, and Ana Cavalcanti. From Circus to JCSP. *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, 3308:320–340, 2004.

[135] Daniel Plagge and Michael Leuschel. Validating Z Specifications Using the ProB Animator and Model Checker. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer, 2007.

[136] GD Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 1981.

[137] V. Raju, L. Rong, and G. S. Stiles. {A}utomatic {C}onversion of {CSP} to {CTJ}, {JCSP}, and {CCSP}. In Jan F Broenink and Gerald H Hilderink, editors, *{C}ommunicating {P}rocess {A}rchitectures 2003*, pages 63–81, sep 2003.

[138] Ken Robinson. The B Method and the B Toolkit, 1997.

[139] Alexander Romanovsky and Martyn Thomas. *Industrial Deployment of System Engineering Methods*, volume 9783642331. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[140] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.

[141] A. W. Roscoe. *Understanding Concurrent Systems*, volume 42 of *Texts in Computer Science.* Springer London, London, 2010.

[142] A. W. Roscoe, P Gardiner, M Goldsmith, J Hulance, D Jackson, and J Scattergood. Hierarchical compression for model-checking CSP or how to check 1020 dining philosophers for deadlock. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.

[143] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–53, 1996.

[144] RTCA Special Committee 190 and EUROCAE Working Group 52. Errata DO-178B. Software considerations in airborne systems and equipment certification, 1999.

[145] Mark Saaltink, Irwin Meisels, and Mark Saaltink. The Z/EVES Reference Manual (for version 1.5). *Reference manual, ORA Canada*, pages 72–85, 1997.

[146] Bryan Scattergood. The semantics and implementation of machine-readable CSP. pages 1–179, 1998.

[147] Nan C Schaller, Gerald H Hilderink, and Peter H Welch. Using Java for Parallel Computing: JCSP versus CTJ, a Comparison. In *Communicating Process Architectures 2000*, pages 205–226. IOS Press, 2000.

[148] Klaus Schneider. *Verification of Reactive Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[149] Steve Schneider. *Concurrent and Real Time Systems: the CSP approach*. Number c. John Wiley, 2010.

[150] Steve Schneider, Helen Treharne, Alistair A. McEwan, and Wilson Ifill. Experiments in Translating CSP||B to Handel-C. *Communicating Process Architectures*, 2008.

[151] Adnan Sherif, Ana Cavalcanti, H Jifeng, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, mar 2010.

[152] Adnan Sherif and Jifeng He. Towards a Time Model for Circus. In George Chris, , and Huaikou Miao, editors, *4th International Conference on Formal Engineering Methods*, pages 206,615. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[153] Neeraj Kumar SINGH. Fiabilité et Sûreté des Systèmes Informatiques Critiques. *Mosel (Loria)*, 2011.

[154] Neeraj Kumar Singh. Critical System Development Methodology. *Using Event-B for Critical Device Software Systems*, NA:61–77, 2013.

[155] Neeraj Kumar Singh, Andy Wellings, and Ana L. C. Cavalcanti. The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES '12*, page 62, 2012.

[156] Graeme Smith and John Derrick. Refinement and Verification of Concurrent Systems Specified in Object-Z and CSP. In *Icfem'97*, pages 293–302. IEEE Computer Society Press, 1997.

[157] J Michael Spivey. The Z notation: a reference manual. International Series in Computer Science, 1992.

[158] Jun Sun, Yang Liu, and Jin Song Dong. Model checking CSP revisited: Introducing a process analysis toolkit. In T. Margaria and B. Steffen, editors, *Communications in Computer and Information Science*, volume 17 CCIS, pages 307–322, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[159] K Taguchi and K Araki. The state-based CCS semantics for concurrent Z specification. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, page 283, Washington, DC, USA, 1997. IEEE Computer Society.

[160] Gerwin Klein Tobias Nipkow, Tobias Nipkow, Gerwin Klein, With Isabelle, and Gerwin Klein Tobias Nipkow. Concrete Semantics. pages 1–2, 2016.

[161] Helen Treharne and Steve Schneider. Using a Process Algebra to Control B OPERATIONS. In Araki Keijiro, Andy Galloway, and Taguchi Kenji, editors, *IFM99 1st International Conference on Integrated Formal Methods*, number CSD-TR-99-01, pages 437–457, London, 1999. Springer London.

[162] Mark Utting. Jaza User Manual and Tutorial, jun 2005.

[163] Marcel Verhoef. On the Use of VDM++ for Specifying Real-Time Systems. *Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop, Newcastle, July 2005*, pages 26–43, 2006.

[164] Kun Wei. New Circus Time. (April), 2013.

[165] Peter H. Welch. Process Oriented Design for Java–Concurrency for All. *Pdpta 2000*, 1:51–57, 2009.

[166] Peter H. Welch. Life of occam-Pi. *Communicating Process Architectures 2013*, pages 293–318, 2013.

[167] Peter H. Welch and Frederick R. M. Barnes. Communicating Mobile Processes: introducing occam-pi. In *Communicating Sequential Processes: The First 25 Years*, 2005.

[168] Peter H. Welch and Neil C.C. Brown. Communicationing Sequential Processes for Java (JCSP), 1999.

[169] Peter H. Welch, Neil C.C. Brown, J S Moore, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. *Communicating Process Architectures 2007*, pages 349–369, 2007.

[170] J. C. P. Woodcock and Ana Cavalcanti. The Semantics of Circus. In *Zb 2002: Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users Grenoble*, 2002.

[171] J. C. P. Woodcock, Ana L. C. Cavalcanti, and Leonardo Freitas. Operational Semantics for model checking Circus. *Lecture Notes in Computer Science*, 3582:237–252, 2005.

[172] J. C. P. Woodcock and Jim Davies. *Using Z, Specification, Refinement, and Proof.* Prentice Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[173] Letu Yang and Michael Poppleton. Automatic Translation from Combined \emph{B} and {CSP} Specification to Java Programs. In *B 2007: Formal Specification and Development in B, 7th International Conference of {B} Users, Besançon, France, January 17-19, 2007, Proceedings*, pages 64–78, 2007.

[174] Kangfeng Ye. *Model Checking of State-Rich Formalisms.* PhD thesis, University of York, 2016.

[175] Kangfeng Ye and J. C. P. Woodcock. Model checking of state-rich formalism Circus by linking to CSP||B. *International Journal on Software Tools for Technology Transfer*, 19(1):73–96, 2017.

[176] Frank Zeyda and Ana Cavalcanti. Encoding Circus programs in ProofPower-Z. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5713 LNCS:218–237, 2010.

[177] Frank Zeyda and Ana Cavalcanti. Mechanical reasoning about families of UTP theories. In *Science of Computer Programming*, volume 77, pages 444–479, 2012.

[178] Weiyang Zhou and G. S. Stiles. The automated serialization of concurrent CSP scripts using Mathematica. In *Communicating Process Architectures Conference, Enschede, the Netherlands*, 2000.

Appendix A

Circus AST

A.1 *Circus* and Z Paragraphs

$$
\begin{array}{lcl}
ZPara & ::= & \mathbf{channel}\ CDecl \\
 & | & \mathbf{channelset}\ N == CSExp \\
 & | & ProcDecl
\end{array}
$$

A.2 *Circus* Channel Declaration – *CDecl*

$$
\begin{array}{lcl}
CDecl & ::= & n \\
 & | & n : Exp \\
 & | & n[e] : Exp
\end{array}
$$

A.3 *Circus* Process – *ProcDecl*

$$
\begin{array}{lcl}
ProcDecl & ::= & \mathbf{process}\ N \mathrel{\widehat=} ProcDef \\
 & | & \mathbf{process}\ N(N^{+}) \mathrel{\widehat=} ProcDef \\
 & | & \mathbf{process}\ N[N^{+}] \mathrel{\widehat=} ProcDef
\end{array}
$$

A.4 *Circus* Process – *ProcessDef*

$$
\begin{array}{lcl}
ProcessDef & ::= & Decl \bullet ProcDef \\
 & | & Decl \odot ProcDef \\
 & | & Proc
\end{array}
$$

A.5 *Circus* Process – $CProc$

$$
\begin{array}{lcl}
CProc & ::= & N \\
& | & N[Exp^+] \\
& | & N(Exp^+) \\
& | & N\lfloor Exp^+ \rfloor \\
& | & \mathop{;} Decl \bullet CProc \\
& | & \Box\, Decl \bullet CProc \\
& | & \sqcap\, Decl \bullet CProc \\
& | & [\![CSExp]\!]\, Decl \bullet CProc \\
& | & |||\, Decl \bullet CProc \\
& | & CProc \setminus CSExp \\
& | & CProc \Box CProc \\
& | & CProc \sqcap CProc \\
& | & CProc [\![CSExp]\!] CProc \\
& | & CProc ||| CProc \\
& | & CProc \,;\, CProc \\
& | & (Decl \bullet ProcDef)(Exp^+) \\
& | & \mathbf{begin}\ PPar* \\
& & \quad \mathbf{state}\ SchemaExpPPar* \\
& & \quad \bullet\ CAction \\
& & \mathbf{end} \\
& | & \mathbf{begin}\ PPar* \\
& & \quad \bullet\ CAction \\
& & \mathbf{end}
\end{array}
$$

A.6 Process paragraphs – $PPar$

$$
\begin{array}{lcl}
PPar & ::= & ZPara \\
& | & N \mathrel{\widehat=} ParAction \\
& | & NSExp
\end{array}
$$

A.7 Parametrised Actions – $ParAction$

$$
\begin{array}{lcl}
ParAction & ::= & CAction \\
& | & Decl \bullet ParAction
\end{array}
$$

A.8 *Circus* Actions – *CAction*

$$
\begin{array}{lll}
CAction & ::= & (S) \\
 & | & Name \\
 & | & Skip \\
 & | & Stop \\
 & | & Chaos \\
 & | & Comm \rightarrow CAction \\
 & | & CCommand \\
 & | & (Pred) \mathbin{\&} CAction \\
 & | & CAction \; ; \; CAction \\
 & | & CAction \Box CAction \\
 & | & CAction \sqcap CAction \\
 & | & CAction \llbracket NSExp \mid CSExp \mid NSExp \rrbracket CAction \\
 & | & CAction \llbracket CSExp \rrbracket CAction \\
 & | & CAction \interleave CAction \\
 & | & CAction \setminus CSExp \\
 & | & CAction(Exp^{+}) \\
 & | & CAction[x/y, z/n] \\
 & | & \mu N \bullet CAction \\
 & | & (Decl \bullet CAction)(ZName) \\
 & | & \mathop{;} Decl \bullet CAction \\
 & | & \Box\, Decl \bullet CAction \\
 & | & \sqcap\, Decl \bullet CAction \\
 & | & \llbracket CSExp \rrbracket\, Decl \bullet \llbracket NSExp \rrbracket\, CAction \\
 & | & \llbracket CSExp \rrbracket\, Decl \bullet CAction \\
 & | & \interleave\, Decl \bullet \lVert\!\llbracket NSExp \rrbracket\!\rVert\, CAction \\
 & | & \interleave\, Decl \bullet CAction
\end{array}
$$

A.9 *Circus* Communication

$$
\begin{array}{lll}
Comm & ::= & NCParameter* \\
 & | & N[Exp^{+}]CParameter*
\end{array}
$$

A.10 *Circus* Communication – *CParameter*

$$
\begin{array}{lll}
CParameter & ::= & ?N \\
 & | & ?N : Pred \\
 & | & !Exp \\
 & | & .Exp
\end{array}
$$

A.11 *Circus* Commands – *CCommand*

$$
\begin{array}{lll}
CCommand & ::= & N^{+} := Exp^{+} \\
 & | & \mathbf{if}\ GActions\ \mathbf{fi} \\
 & | & \mathbf{var}\ Decl \bullet CAction \\
 & | & \mathbf{val}\ Decl \bullet CAction \\
 & | & \mathbf{res}\ Decl \bullet CAction \\
 & | & \mathbf{vres}\ Decl \bullet CAction \\
 & | & N^{+} : [Pred, Pred] \\
 & | & N^{+} : [Pred] \\
 & | & :[Pred, Pred] \\
 & | & :[Pred] \\
 & | & \{Pred\} \\
 & | & [Pred]
\end{array}
$$

A.12 *Circus* Guards – *CGActions*

$$
\begin{array}{lll}
CGActions & ::= & Pred \longrightarrow CAction \\
 & | & CGActions \mathbin{[\!]} GActions
\end{array}
$$

A.13 *Circus* Renaming – *CReplace*

$$
\begin{array}{lll}
CCommand & ::= & CAction[N^{+}/M^{+}] \\
 & | & \mathbf{if}\ CAction[N^{+} := M^{+}]
\end{array}
$$

Appendix B

Omega Mapping Functions

B.1 *Circus* and Z Paragraphs

$$
\begin{array}{rcl}
\Omega_{ZPara}(\mathbf{channel}\ CDecl) & \widehat{=} & \mathbf{channel}\ CDecl \\
\Omega_{ZPara}(\mathbf{channelset}\ N == CSExp) & \widehat{=} & \mathbf{channelset}\ N == CSExp \\
\Omega_{ZPara}(Pr) & \widehat{=} & \Omega_{ProcDecl}(Pr)
\end{array}
$$

B.2 Mapping *Circus* Processes Declaration

B.2.1 *Circus* Process − *ProcDecl*

$$
\begin{array}{rcl}
\Omega_{ProcDecl}(\mathbf{process}\ N \widehat{=} P) & \widehat{=} & \mathbf{process}\ N \widehat{=} \Omega_{ProcDecl}(P) \\
\Omega_{ProcDecl}(\mathbf{process}\ N(N^{+}) \widehat{=} P) & \widehat{=} & \mathbf{process}\ N(N^{+}) \widehat{=} \Omega_{ProcDecl}(P) \\
\Omega_{ProcDecl}(\mathbf{process}\ N[N^{+}] \widehat{=} P) & \widehat{=} & \mathbf{process}\ N[N^{+}] \widehat{=} \Omega_{ProcDecl}(P)
\end{array}
$$

B.2.2 *Circus* Process − *ProcessDef*

$$
\begin{array}{rcl}
\Omega_{ProcessDef}(Decl \bullet PD) & \widehat{=} & (Decl \bullet \Omega_{ProcessDef}(PD)) \\
\Omega_{ProcessDef}(Decl \odot PD) & \widehat{=} & Decl \odot \Omega_{ProcessDef}(PD) \\
\Omega_{ProcessDef}(Proc) & \widehat{=} & \Omega_{CProc}(Proc)
\end{array}
$$

B.2.3 *Circus* Process – *CProc*

$$
\begin{array}{rcl}
\Omega_{CProc}(N) & \widehat{=} & N \\
\Omega_{CProc}(N[Exp^{+}]) & \widehat{=} & N[Exp^{+}] \\
\Omega_{CProc}(N(Exp^{+})) & \widehat{=} & N(Exp^{+}) \\
\Omega_{CProc}(N\lfloor Exp^{+}\rfloor) & \widehat{=} & N\lfloor Exp^{+}\rfloor \\
\Omega_{CProc}(\mathbin{;}\ Decl \bullet Proc) & \widehat{=} & \mathbin{;}\ Decl \bullet \Omega_{CProc}(Proc) \\
\Omega_{CProc}(\Box\, Decl \bullet Proc) & \widehat{=} & \Box\, Decl \bullet \Omega_{CProc}(Proc) \\
\Omega_{CProc}(\sqcap\, Decl \bullet Proc) & \widehat{=} & \sqcap\, Decl \bullet \Omega_{CProc}(Proc) \\
\Omega_{CProc}(\llbracket CSExp \rrbracket\, Decl \bullet Proc) & \widehat{=} & \llbracket CSExp \rrbracket\, Decl \bullet \Omega_{CProc}(Proc) \\
\Omega_{CProc}(\vert\vert\vert\, Decl \bullet Proc) & \widehat{=} & \vert\vert\vert\, Decl \bullet \Omega_{CProc}(Proc) \\
\Omega_{CProc}(Proc \setminus CSExp) & \widehat{=} & \Omega_{CProc}(Proc) \setminus CSExp \\
\Omega_{CProc}(Proc \Box Proc) & \widehat{=} & \Omega_{CProc}(Proc) \Box \Omega_{CProc}(Proc) \\
\Omega_{CProc}(Proc \sqcap Proc) & \widehat{=} & \Omega_{CProc}(Proc) \sqcap \Omega_{CProc}(Proc) \\
\Omega_{CProc}(Proc \llbracket CSExp \rrbracket Proc) & \widehat{=} & \Omega_{CProc}(Proc) \llbracket CSExp \rrbracket \Omega_{CProc}(Proc) \\
\Omega(Proc \vert\vert\vert Proc) & \widehat{=} & \Omega_{CProc}(Proc) \vert\vert\vert \Omega_{CProc}(Proc) \\
\Omega_{CProc}(Proc \mathbin{;} Proc) & \widehat{=} & \Omega_{CProc}(Proc) \mathbin{;} \Omega_{CProc}(Proc) \\
\Omega_{CProc}((Decl \bullet ProcDef)(Exp^{+})) & \widehat{=} & ((Decl \bullet \Omega_{ProcessDef}(ProcDef))(Exp^{+}))
\end{array}
$$

The translation of state-rich *Circus* process is detailed in Section 4.

B.3 Mapping Parameterised *Circus* Actions

$$
\begin{array}{rcl}
\Omega_{PPar}(ZPara) & \widehat{=} & \Omega_{ZSchema}(ZPara) \\
\Omega_{PPar}(N \widehat{=} PA) & \widehat{=} & N \widehat{=} \\
\Omega_{ParAction}(PA) & & \\
\Omega_{PPar}(NSExp) & \widehat{=} & NSExp
\end{array}
$$

B.3.1 Parametrised Actions – *ParAction*

$$
\begin{array}{rcl}
\Omega_{ParAction}(Decl \bullet PA) & \widehat{=} & Decl \bullet \Omega_{ParAction}(PA) \\
\Omega_{ParAction}(CA) & \widehat{=} & \Omega_{CAction}(CA)
\end{array}
$$

B.3.2 Stateless *Circus* - Actions - Derived from Deliverable 24.1 [132]

Ω_A 1

$\Omega_A(\text{Skip}) \widehat{=} \text{Skip}$
$\Omega_A(\text{Stop}) \widehat{=} \text{Stop}$
$\Omega_A(\text{Chaos}) \widehat{=} \text{Chaos}$

Ω_A 2

$\Omega_A(c \rightarrow A) \widehat{=} c \rightarrow \Omega_A(A)$

Ω_A 3

$$
\Omega_A(c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \widehat{=}
\left(\begin{array}{l}
mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\
mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\
c.e(vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \rightarrow \Omega'_A(A)
\end{array}\right)
$$

where

$$FV(e) = (v_0, \ldots, v_n, l_0, \ldots, l_m)$$

Ω_A **4**

Included by Artur - An input carrying a value named with a state variable is defined as an assignment to that, but as assignments are not allowed, we directly make a mset with that value. Therefore, suppose $v_n \in \mathit{StateVariables}$, we have the following translation rule.

$$\Omega_A(c?v_n \to A) \mathrel{\widehat=} \left(\ c?vv_n \to mset.v_n!vv_n(vv_0, ..., vv_n, vl_0, ..., vl_m) \to \Omega_A(A)\ \right)$$

where

$$v_n \in (v_0, \ldots, v_n, l_0, \ldots, l_m)$$

Ω_A **5**

$$\Omega_A(c?x : P(x, v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) \mathrel{\widehat=} \left(\begin{array}{l} mget.v_0?vv_0 \to \ldots \to mget.v_n?vv_n \to \\ mget.l_0?vl_0 \to \ldots \to mget.l_m?vl_m \to \\ c?x : P(x, vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \to \Omega'_A(A) \end{array}\right)$$

where

$$x \in wrtV(A)$$

Ω_A **6**

$$\Omega_A(c?x \to A) \mathrel{\widehat=} c?x \to \Omega'_A(A)$$

Ω_A **7**

$$\Omega_A(A_1 \interleave A_2) \mathrel{\widehat=} \Omega_A(A_1) \interleave \Omega_A(A_2)$$

Ω_A **8**

$$\Omega_A(c!e(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) \mathrel{\widehat=} c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A$$

Ω_A **9**

$$\Omega_A(g(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) \mathrel{\widehat=} \left(\begin{array}{l} mget.v_0?vv_0 \to \ldots \to mget.v_n?vv_n \to \\ mget.l_0?vl_0 \to \ldots \to mget.l_m?vl_m \to \\ g(vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \,\&\, \Omega'_A(A) \end{array}\right)$$

Ω_A **10**

$$\Omega_A(A_1 \,;\; A_2) \mathrel{\widehat=} \Omega_A(A_1) \,;\; \Omega_A(A_2)$$

Ω_A **11**

$$\Omega_A(A_1 \sqcap A_2) \mathrel{\widehat=} \Omega_A(A_1) \sqcap \Omega_A(A_2)$$

Ω_A **12**

$$\Omega_A(A_1 \Box A_2) \mathrel{\widehat=} \left(\begin{array}{l} mget.v_0?vv_0 \to \ldots \to mget.v_n?vv_n \to \\ mget.l_0?vl_0 \to \ldots \to mget.l_m?vl_m \to \\ (\Omega'_A(A_1) \Box \Omega'_A(A_2)) \end{array}\right)$$

Ω_A **13**

The definition of parallel composition (and interleaving), as defined in the D24.1, has a *MemoryMerge*, MRG_I and *Merge* components and channel sets. Whilst translating them into CSP, the tool would rather expand their definition.

$$\Omega_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2) \mathrel{\widehat=} \Gamma_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$

Ω_A **14**

$$\Omega_A(\mathop{;}\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1)\ ;\ \ldots ;\ A(v_n))$$

Ω_A **15**

$$\Omega_A(\Box\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1) \Box \ldots \Box A(v_n))$$

Ω_A **16**

$$\Omega_A(\sqcap\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega_A(A(v_1) \sqcap \ldots \sqcap A(v_n))$$

Ω_A **17**

$$\Omega_A(\llbracket cs \rrbracket\, x : \langle v_1, ..., v_n \rangle \bullet \llbracket ns(x) \rrbracket\, A(x)) \mathrel{\widehat=} \left(\begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}\rrbracket \\ \left(\ldots \left(\begin{array}{l} \Omega_A(A(v_n - 1)) \\ \llbracket ns(v_n - 1) \mid cs \mid ns(v_n)\rrbracket \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

Ω_A **18**

$$\Omega_A(\mathbf{val}\, Decl \bullet P) \mathrel{\widehat=} \mathbf{val}\, Decl \bullet \Omega_A(P)$$

Ω_A **19**

$$\Omega_A \left(x_0, \ldots, x_n := e_0 \left(\begin{array}{l} v_0, ..., v_n, \\ l_0, ..., l_m \end{array} \right), \ldots, e_n \left(\begin{array}{l} v_0, ..., v_n, \\ l_0, ..., l_m \end{array} \right) \right) \mathrel{\widehat=} \left(\begin{array}{l} mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\ mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\ mset.x_0!e_0(vv_0, ..., vv_n, vl_0, ..., vl_m) \rightarrow \ldots \rightarrow \\ mset.x_n!e_n(vv_0, ..., vv_n, vl_0, ..., vl_m) \rightarrow \mathrm{Skip} \end{array} \right)$$

Ω_A **20**

$$\Omega_A(A \setminus cs) \mathrel{\widehat=} \Omega_A(A) \setminus cs$$

Ω_A **21**

$$\Omega_A \left(\begin{array}{l} \mathbf{if}\ g_0(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A_0 \\ \quad [\!] \ldots \\ \quad [\!] g_n(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow A_n \\ \mathbf{fi} \end{array} \right) \mathrel{\widehat=} \left(\begin{array}{l} mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow \\ mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\ \mathbf{if}\ g_0(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A_0) \\ \quad [\!] \ldots \\ \quad [\!] g_n(v_0, ..., v_n, l_0, ..., l_m) \longrightarrow \Omega'_A(A_n) \\ \mathbf{fi} \end{array} \right)$$

Ω_A **22**

$$\Omega_A(\mu X \bullet A(X)) \mathrel{\widehat=} \mu X \bullet \Omega_A(A(X))$$

Ω_A **23**

$$\Omega_A(\interleave x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \left(\begin{array}{l} A(v_1) \\ [\![ns(v_1) \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}]\!] \\ \left(\ldots \left(\begin{array}{l} \Omega_A(A(v_n - 1)) \\ [\![ns(v_n - 1) \mid ns(v_n)]\!] \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

Ω_A **24**

$$\Omega_A(A[old_1, ..., old_n := new_1, ..., new_n) \mathrel{\widehat=} A[new_1, ..., new_n / old_1, ..., old_n)$$

B.3.3 Definitions of Ω'_A

Ω'_A **1**

$$\Omega'_A(\text{Skip}) \mathrel{\widehat=} \text{Skip}$$
$$\Omega'_A(\text{Stop}) \mathrel{\widehat=} \text{Stop}$$
$$\Omega'_A(\text{Chaos}) \mathrel{\widehat=} \text{Chaos}$$

Ω'_A **2**

$$\Omega'_A(c \to A) \mathrel{\widehat=} c \to \Omega'_A(A)$$

Ω'_A **3**

$$\Omega'_A(c.e \to A) \mathrel{\widehat=} c(vv_0, ..., vv_n, vl_0, ..., vl_m) \to \Omega'_A(A)$$

Ω'_A **4**

$$\Omega'_A(c!e \to A) \mathrel{\widehat=} c.e \to A$$

Ω'_A **5**

$$\Omega'_A(g \to A) \mathrel{\widehat=} g \to \Omega'_A(A)$$

Ω'_A **6**

$$\Omega'_A(c?x \to A) \mathrel{\widehat=} c?x \to \Omega'_A(A)$$

Ω'_A **7**

$$\Omega'_A(c?x : P \to A) \mathrel{\widehat=} c?x : P \to \Omega'_A(A)$$

where

$$x \in wrtV(A)$$

Ω'_A **8**

$$\Omega'_A(A_1 \ ; \ A_2) \mathrel{\widehat=} \Omega'_A(A_1) \ ; \ \Omega_A(A_2)$$

Ω'_A **9**

$$\Omega'_A(A_1 \sqcap A_2) \mathrel{\widehat=} \Omega'_A(A_1) \sqcap \Omega'_A(A_2)$$

Ω'_A **10**

$$\Omega'_A(A_1 \Box A_2) \mathrel{\widehat=} (\Omega'_A(A_1) \Box \Omega'_A(A_2))$$

Ω'_A **11**

$$\Omega'_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2) \mathrel{\widehat=} \Gamma_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2)$$

Ω'_A **12**

$$\Omega'_A(\mathop{;}\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega'_A(A(v_1)\ ;\ \ldots ;\ A(v_n))$$

Ω'_A **13**

$$\Omega'_A(\Box\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega'_A(A(v_1) \Box \ldots \Box A(v_n))$$

Ω'_A **14**

$$\Omega'_A(\sqcap\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Omega'_A(A(v_1) \sqcap \ldots \sqcap A(v_n))$$

Ω'_A **15**

$$\Omega'_A(\llbracket cs \rrbracket\, x : \langle v_1, ..., v_n \rangle \bullet \llbracket ns(x) \rrbracket\, A(x)) \mathrel{\widehat=} \left(\begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}\rrbracket \\ \left(\begin{array}{l} \ldots \\ \left(\begin{array}{l} \Omega'_A(A(v_n - 1)) \\ \llbracket ns(v_n - 1) \mid cs \mid ns(v_n)\rrbracket \\ A(v_n) \end{array} \right) \end{array} \right) \end{array} \right)$$

Ω'_A **16**

$$\Omega'_A(\mathbf{val}\, Decl \bullet P) \mathrel{\widehat=} \mathbf{val}\, Decl \bullet \Omega'_A(P)$$

Ω'_A **17**

$$\Omega'_A\left(\ x_0, \ldots, x_n := e_0, \ldots, e_n\ \right) \mathrel{\widehat=} mset.x_0!e_0 \to \ldots \to mset.x_n!e_n \to \mathrm{Skip}$$

Ω'_A **18**

$$\Omega'_A(A \setminus cs) \mathrel{\widehat=} \Omega'_A(A) \setminus cs$$

Ω'_A **19**

$$\Omega'_A \left(\begin{array}{l} \mathbf{if}\ g_0 \longrightarrow A_0 \\ \quad [] \ldots \\ \quad [] g_n \longrightarrow A_n \\ \mathbf{fi} \end{array} \right) \mathrel{\widehat=} \left(\begin{array}{l} \mathbf{if}\ g_0 \longrightarrow \Omega'_A(A_0) \\ \quad [] \ldots \\ \quad [] g_n \longrightarrow \Omega'_A(A_n) \\ \mathbf{fi} \end{array} \right)$$

Ω'_A **20**

$$\Omega'_A(\mu X \bullet A(X)) \mathrel{\widehat=} \mu X \bullet \Omega'_A(A(X))$$

Ω'_A **21**

$$\Omega'_A(\left|\left|\right|\right|\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \left(\begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}\rrbracket \\ \left(\ldots \left(\begin{array}{l} \Omega'_A(A(v_n - 1)) \\ \llbracket ns(v_n - 1) \mid ns(v_n)\rrbracket \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

Ω'_A **22**

$$\Omega'_A(A[old_1, ..., old_n := new_1, ..., new_n) \mathrel{\widehat=} A[new_1, ..., new_n / old_1, ..., old_n)$$

B.4 Γ functions

This section is a novelty of the approach. It describes all the mapping functions that runs within the scope of parallel actions. As described in Section **??**, it creates an intermediate using the *MemoryMerge* action, with local *lgets* and *lsets* and then, should a *lterminate* signal arises, all the values of such intermediate state are propagated to the *Memory* action through *mget* and *mset*.

B.4.1 Stateless *Circus* - Actions

Γ_A **1**

$$\Gamma_A(\text{Skip}) \mathrel{\widehat=} \text{Skip}$$
$$\Gamma_A(\text{Stop}) \mathrel{\widehat=} \text{Stop}$$
$$\Gamma_A(\text{Chaos}) \mathrel{\widehat=} \text{Chaos}$$

Γ_A **2**

$$\Gamma_A(c \rightarrow A) \mathrel{\widehat=} c \rightarrow \Gamma_A(A)$$

Γ_A **3**

$$\Gamma_A(c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \mathrel{\widehat=} \left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ c.e(vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \rightarrow \Gamma'_A(A) \end{array}\right)$$

where

$$FV(e) = (v_0, \ldots, v_n, l_0, \ldots, l_m)$$

Γ_A **4**

$$\Gamma_A(c!e(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \mathrel{\widehat=} c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A$$

Γ_A **5**

$$\Gamma_A(g(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \mathrel{\widehat=} \left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ g(vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \; \& \; \Gamma'_A(A) \end{array}\right)$$

Γ_A **6**

$$\Gamma_A(c?x : P(x, v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \mathrel{\widehat=} \left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ c?x : P(x, vv_0, \ldots, vv_n, vl_0, \ldots, vl_m) \rightarrow \Gamma'_A(A) \end{array}\right)$$

where

$$x \in wrtV(A)$$

Γ_A **7**

$$\Gamma_A(A_1 \; ; \; A_2) \mathrel{\widehat=} \Gamma_A(A_1) \; ; \; \Gamma_A(A_2)$$

Γ_A **8**

$$\Gamma_A(A_1 \sqcap A_2) \mathrel{\widehat=} \Gamma_A(A_1) \sqcap \Gamma_A(A_2)$$

Γ_A **9**

$$\Gamma_A(A_1 \Box A_2) \mathrel{\widehat=} \left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ (\Gamma'_A(A_1) \Box \Gamma'_A(A_2)) \end{array}\right)$$

Γ_A **10**

$$\begin{array}{l} \Gamma_A(A1 \llbracket ns1 \mid cs \mid ns2 \rrbracket A2) \mathrel{\widehat=} \\ \left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ \left(\begin{array}{l} \left(\begin{array}{l} (\ \Gamma'_A(A_1)\ ;\ terminate \rightarrow \mathrm{Skip}\ ) \\ \llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\ MemoryMerge(\{v0 \mapsto vv0, \ldots\}, LEFT) \end{array}\right) \setminus MEM_I \\ \llbracket \{\} \mid cs \mid \{\} \rrbracket \\ \left(\begin{array}{l} (\ \Gamma'_A(A_2)\ ;\ terminate \rightarrow \mathrm{Skip}\ ) \\ \llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\ MemoryMerge(\{v0 \mapsto vv0, \ldots\}, RIGHT) \end{array}\right) \setminus MEM_I \end{array}\right) \end{array}\right) \end{array}$$

The definition of parallel composition (and interleaving), as defined in the D24.1, has a *MemoryMerge*, MRG_I and *Merge* components and channel sets. Whilst translating them into CSP, the tool would rather expand their definition. Γ_A **11**

$$\Gamma_A(\mathbin{;}\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Gamma_A(A(v_1)\ ;\ \ldots\ ;\ A(v_n))$$

Γ_A **12**

$$\Gamma_A(\Box\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Gamma_A(A(v_1) \Box \ldots \Box A(v_n))$$

Γ_A **13**

$$\Gamma_A(\sqcap\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Gamma_A(A(v_1) \sqcap \ldots \sqcap A(v_n))$$

Γ_A **14**

$$\begin{array}{l} \Gamma_A(\llbracket cs \rrbracket\ x : \langle v_1, ..., v_n \rangle \bullet \llbracket ns(x) \rrbracket\ A(x)) \mathrel{\widehat=} \\ \left(\begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\} \rrbracket \\ \left(\begin{array}{l} \ldots \\ \left(\begin{array}{l} \Gamma_A(A(v_n - 1)) \\ \llbracket ns(v_n - 1) \mid cs \mid ns(v_n) \rrbracket \\ A(v_n) \end{array}\right) \end{array}\right) \end{array}\right) \end{array}$$

Γ_A **15**

$$\Gamma_A(\mathbf{val}\, Decl \bullet P) \mathrel{\widehat=} \mathbf{val}\, Decl \bullet \Gamma_A(P)$$

Γ_A **16**

$$\begin{array}{l} \Gamma_A\left(\ x_0, \ldots, x_n := e_0\left(\begin{array}{l} v_0, ..., v_n, \\ l_0, ..., l_m \end{array}\right), \ldots, e_n\left(\begin{array}{l} v_0, ..., v_n, \\ l_0, ..., l_m \end{array}\right)\ \right) \mathrel{\widehat=} \\ \left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ lset.x_0!e_0(vv_0, ..., vv_n, vl_0, ..., vl_m) \rightarrow \\ ldots \rightarrow \\ lset.x_n!e_n(vv_0, ..., vv_n, vl_0, ..., vl_m) \rightarrow \mathrm{Skip} \end{array}\right) \end{array}$$

Γ_A **17**

$$\Gamma_A(A \setminus cs) \mathrel{\widehat=} \Gamma_A(A) \setminus cs$$

Γ_A **18**

$$\Gamma_A \left(\begin{array}{l} \textbf{if } g_0(v_0,...,v_n,l_0,...,l_m) \longrightarrow A_0 \\ \quad [] \ldots \\ \quad [] g_n(v_0,...,v_n,l_0,...,l_m) \longrightarrow A_n \\ \textbf{fi} \end{array} \right) \mathrel{\widehat=}$$

$$\left(\begin{array}{l} lget.v_0?vv_0 \rightarrow \ldots \rightarrow lget.v_n?vv_n \rightarrow \\ \quad lget.l_0?vl_0 \rightarrow \ldots \rightarrow lget.l_m?vl_m \rightarrow \\ \quad \textbf{if } g_0(v_0,...,v_n,l_0,...,l_m) \longrightarrow \Gamma'_A(A_0) \\ \quad\quad [] \ldots \\ \quad\quad [] g_n(v_0,...,v_n,l_0,...,l_m) \longrightarrow \Gamma'_A(A_n) \\ \quad \textbf{fi} \end{array} \right)$$

Γ_A **19**

$$\Gamma_A(\mu X \bullet A(X)) \mathrel{\widehat=} \mu X \bullet \Gamma_A(A(X))$$

Γ_A **20**

$$\Gamma_A(|||\ x : \langle v_1,...,v_n \rangle \bullet A(x)) \mathrel{\widehat=}$$

$$\left(\begin{array}{l} A(v_1) \\ [\![ns(v_1) \mid \bigcup\{x : \{v_2,...,v_n\} \bullet ns(x)\}]\!] \\ \left(\ldots \left(\begin{array}{l} \Gamma_A(A(v_n - 1)) \\ [\![ns(v_n - 1) \mid ns(v_n)]\!] \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

Γ_A **21**

$$\Gamma_A(A[old_1,...,old_n := new_1,...,new_n) \mathrel{\widehat=} \\ A[new_1,...,new_n/old_1,...,old_n)$$

B.4.2 Definitions of Γ'_A

Γ'_A **1**

$$\Gamma'_A(\text{Skip}) \mathrel{\widehat=} \text{Skip}$$
$$\Gamma'_A(\text{Stop}) \mathrel{\widehat=} \text{Stop}$$
$$\Gamma'_A(\text{Chaos}) \mathrel{\widehat=} \text{Chaos}$$

Γ'_A **2**

$$\Gamma'_A(c \rightarrow A) \mathrel{\widehat=} c \rightarrow \Gamma'_A(A)$$

Γ'_A **3**

$$\Gamma'_A(c.e \rightarrow A) \mathrel{\widehat=} c(vv_0,...,vv_n,vl_0,...,vl_m) \rightarrow \Gamma'_A(A)$$

Γ'_A **4**

$$\Gamma'_A(c!e \rightarrow A) \mathrel{\widehat=} c.e \rightarrow A$$

Γ'_A **5**

$$\Gamma'_A(c?x \rightarrow A) \mathrel{\widehat=} c?x \rightarrow \Gamma'_A(A)$$

Γ'_A **6**

$$\Gamma'_A(c?x : P \rightarrow A) \mathrel{\widehat=} c?x : P \rightarrow \Gamma'_A(A)$$

where

$$x \in wrtV(A)$$

Γ'_A **7**

$$\Gamma'_A(g \,\&\, A) \mathrel{\widehat=} g \,\&\, \Gamma'_A(A)$$

Γ'_A **8**

$$\Gamma'_A(A_1 \,;\; A_2) \mathrel{\widehat=} \Gamma'_A(A_1) \,;\; \Gamma_A(A_2)$$

Γ'_A **9**

$$\Gamma'_A(A_1 \sqcap A_2) \mathrel{\widehat=} \Gamma'_A(A_1) \sqcap \Gamma'_A(A_2)$$

Γ'_A **10**

$$\Gamma'_A(A_1 \Box A_2) \mathrel{\widehat=} (\Gamma'_A(A_1) \Box \Gamma'_A(A_2))$$

Γ'_A **11**

$$\Gamma'_A(A1 \llbracket\, ns1 \mid cs \mid ns2 \,\rrbracket\, A2) \mathrel{\widehat=}$$
$$\left(\begin{array}{l} lget.v_0?vv_0 \to \ldots \to lget.v_n?vv_n \to \\ lget.l_0?vl_0 \to \ldots \to lget.l_m?vl_m \to \\ \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (\;\Gamma'_A(A_1) \,;\; lterminate \to \mathrm{Skip}\;) \\ \llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\ MemoryMerge(\{v0 \mapsto vv0, \ldots\}, LEFT) \end{array}\right) \setminus MEM_I \\ \llbracket \{\} \mid cs \mid \{\} \rrbracket \\ \left(\begin{array}{l} (\;\Gamma'_A(A_2) \,;\; lterminate \to \mathrm{Skip}\;) \\ \llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\ MemoryMerge(\{v0 \mapsto vv0, \ldots\}, RIGHT) \end{array}\right) \setminus MEM_I \end{array}\right) \\ \llbracket \{\} \mid MEM_I \mid \{\} \rrbracket \\ Merge \end{array}\right) \\ \quad \setminus \{\!|\, mleft, mright \,|\!\} \end{array}\right)$$

Γ'_A **12**

$$\Gamma'_A(\,;\; x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Gamma'_A(A(v_1) \,;\; \ldots \,;\; A(v_n))$$

Γ'_A **13**

$$\Gamma'_A(\Box\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Gamma'_A(A(v_1) \Box \ldots \Box A(v_n))$$

Γ'_A **14**

$$\Gamma'_A(\sqcap\, x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=} \Gamma'_A(A(v_1) \sqcap \ldots \sqcap A(v_n))$$

Γ'_A **15**

$$\Gamma'_A(\llbracket cs \,\rrbracket\, x : \langle v_1, ..., v_n \rangle \bullet \llbracket ns(x) \,\rrbracket\, A(x)) \mathrel{\widehat=}$$
$$\left(\begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\} \rrbracket \\ \left(\ldots \left(\begin{array}{l} \Gamma'_A(A(v_n - 1)) \\ \llbracket ns(v_n - 1) \mid cs \mid ns(v_n) \rrbracket \\ A(v_n) \end{array}\right) \right) \end{array}\right)$$

Γ'_A **16**

$$\Gamma'_A(\mathbf{val}\, Decl \bullet P) \mathrel{\widehat=} \mathbf{val}\, Decl \bullet \Gamma'_A(P)$$

Γ'_A **17**

$$\Gamma'_A \left(\; x_0, \ldots, x_n := e_0, \ldots, e_n \; \right) \mathrel{\widehat=} \left(\begin{array}{l} mset.x_0!e_0 \to \\ \ldots \to \\ mset.x_n!e_n \to \mathrm{Skip} \end{array} \right)$$

Γ'_A **18**

$$\Gamma'_A(A \setminus cs) \mathrel{\widehat=} \Gamma'_A(A) \setminus cs$$

Γ'_A **19**

$$\Gamma'_A \left(\begin{array}{l} \mathbf{if}\ g_0 \longrightarrow A_0 \\ \quad [\!] \ldots \\ \quad [\!] g_n \longrightarrow A_n \\ \mathbf{fi} \end{array} \right) \mathrel{\widehat=} \left(\begin{array}{l} \mathbf{if}\ g_0 \longrightarrow \Gamma'_A(A_0) \\ \quad [\!] \ldots \\ \quad [\!]\ g_n \longrightarrow \Gamma'_A(A_n) \\ \mathbf{fi} \end{array} \right)$$

Γ'_A **20**

$$\Gamma'_A(\mu X \bullet A(X)) \mathrel{\widehat=} \mu X \bullet \Gamma'_A(A(X))$$

Γ'_A **21**

$$\Gamma'_A(\mathop{|||}\ x : \langle v_1, ..., v_n \rangle \bullet A(x)) \mathrel{\widehat=}$$
$$\left(\begin{array}{l} A(v_1) \\ [\![ns(v_1) \mid \bigcup\{x : \{v_2, ..., v_n\} \bullet ns(x)\}]\!] \\ \left(\ldots \left(\begin{array}{l} \Gamma'_A(A(v_n - 1)) \\ [\![ns(v_n - 1) \mid ns(v_n)]\!] \\ A(v_n) \end{array} \right) \right) \end{array} \right)$$

Γ'_A **22**

$$\Gamma'_A(A[old_1, ..., old_n := new_1, ..., new_n) \mathrel{\widehat=} A[new_1, ..., new_n / old_1, ..., old_n)$$

Appendix C

Mapping Functions - *Circus* to CSP

Mapping Functions - *Circus* to CSP

C.1 Mapping *Circus* Paragraphs

The following functions are used to map *Circus* Channels into CSP.

Firstly the *UNIVERSE* type is defined as follows.

$\Upsilon_{CircParagraphs}([UNIVERSE]) \mathrel{\widehat=}$
  `datatype UNIVERSE =` $MK_{Universe}(ulst)$
  $MK_{subtype}(fulst)$
  $MK_{value}(fulst)$
  $MK_{type}(fulst)$

provided

- $\delta(\varnothing) \notin spec$ – There is at least one element in the δ mapping.
- $spec$ is the whole specification
- $ulst = DEF_{Universe}(spec)$ – list of mappings from variables to types, from δ function
- $fulst = GET_{Types}(spec)$ – list of all types from δ function

Then, we define the translation for the channels and sets of the *Memory* model.

$\Upsilon_{CircParagraphs}(\textbf{channel}\, mget, mset : NAME \times UNIVERSE) \mathrel{\widehat=}$
  $\Upsilon_{CDecl}(\textbf{channel}\, mget, mset : NAME \times UNIVERSE)$
$\Upsilon_{CircParagraphs}(\textbf{channel}\, terminate) \mathrel{\widehat=}$
  $\Upsilon_{CDecl}(\textbf{channel}\, terminate)$
$\Upsilon_{CircParagraphs}(\textbf{channelset}\, MEMI == \{\!| mget, mset, terminate |\!\}) \mathrel{\widehat=}$
  `MEM_I =`$\Upsilon_{CSExp}(\{\!| mget, mset, terminate |\!\})$
$\Upsilon_{CircParagraphs}(BINDING \mathrel{\widehat=} NAME \times \mathbb{U}) \mathrel{\widehat=}$
  `BINDINGS_`T_1`= {set(b) | b <- set(distCartProd(NAMES_VALUES_`T_1`))}`
  ...
  `BINDINGS_`T_n`= {set(b) | b <- set(distCartProd(NAMES_VALUES_`T_n`))}`

Finally, we define the translation of the remainder constructs.

$\Upsilon_{CircParagraphs}(N ::= a_1 \mid \ldots \mid a_n) \mathrel{\widehat=}$ `datatype N =` $\Upsilon_{ZBranch_list}(a_1 \mid \ldots \mid a_n)$
$\Upsilon_{CircParagraphs}(P) \mathrel{\widehat=} \Upsilon_{ProcDecl}(P)$
$\Upsilon_{CircParagraphs}(\textbf{channel}\, c) \mathrel{\widehat=} \Upsilon_{CDecl}(c)$
$\Upsilon_{CircParagraphs}(\textbf{channelset}\, CN == CS) \mathrel{\widehat=}$ `CN =` $\Upsilon_{CSExp}(CS)$
$\Upsilon_{CircParagraphs}(N == expr) \mathrel{\widehat=}$ `N =` Υ_{ZExpr}

C.1.1 Mapping *Circus* Channels

This set of mapping functions will translate the declaration of channels from *Circus* into CSP_M. Although, generic channels are not yet available.

$$\Upsilon_{CDecl}(\mathbf{channel}\, a) \mathrel{\widehat=} \texttt{channel a}$$
$$\Upsilon_{CDecl}(\mathbf{channel}\, a : T) \mathrel{\widehat=} \texttt{channel a : T}$$

A channel declaration can be either of form $CChan$ or $CChanDecl$. For $CChan$, we can have **channel** $terminate$, whereas for $CChanDecl$, **channel** $mget : NAME \times BINDING$. Thus, we filter both cases into $x1$ for $CChan$ and $x2$ for $CChanDecl$, and then, display them accordingly.

C.1.2 Mapping Function for Channel Set Expressions

$$\Upsilon_{CSExp}(\{\!|\, xs \,|\!\}) \mathrel{\widehat=} \texttt{\{|}\ \Upsilon_{CSExp_get_cs}(xs)\texttt{|\}}$$
$$\Upsilon_{CSExp}(CS1 \setminus CS2) \mathrel{\widehat=} \texttt{diff(}\Upsilon_{CSExp}(CS1)\texttt{,}\Upsilon_{CSExp}(CS2)\texttt{)}$$
$$\Upsilon_{CSExp}(CS1 \cup CS2) \mathrel{\widehat=} \texttt{union(}\Upsilon_{CSExp}(CS1)\texttt{,}\Upsilon_{CSExp}(CS2)\texttt{)}$$
$$\Upsilon_{CSExp}(CS1 \cap CS2) \mathrel{\widehat=} \texttt{inter(}\Upsilon_{CSExp}(CS1)\texttt{,}\Upsilon_{CSExp}(CS2)\texttt{)}$$
$$\Upsilon_{CSExp}(\{\!|\ |\!\}) \mathrel{\widehat=} \texttt{\{\}}$$
$$\Upsilon_{CSExp}(CS) \mathrel{\widehat=} \texttt{CS}$$

provided

- CS, $CS1$ and $CS2$ are channel sets, $CSExp$.

C.1.3 Mapping *Circus* Processes Declaration

This is the translation rules for $ProcDecl$. Up to the date, we don't have a translation rule for generic processes.

$$\Upsilon_{ProcDecl}(\mathbf{process}\, P \mathrel{\widehat=} ProcDef) \mathrel{\widehat=} \texttt{P}\Upsilon_{ProcessDef}(PD)$$

provided P is the name of a *Circus* process.

C.1.4 Mapping *Circus* Processes Definition

$$\Upsilon_{ProcessDef}(Proc) \mathrel{\widehat=} \texttt{=}\ \Upsilon_{CProc}(Proc)$$
$$\Upsilon_{ProcessDef}(Decl \bullet Proc) \mathrel{\widehat=} \texttt{(}\Upsilon_{ZGenFilt_list}\texttt{) =}\ \Upsilon_{CProc}(Proc)$$

provided

- $Proc$ is the process content.
- $Decl$ are the local variables for the process $Proc$ environment.

C.1.5 Mapping *Circus* Processes

In this section, we list all the mapping functions for the possible behaviours of a *Circus* process. Note that $CGenProc\ (N[Exp^{+}])$, $CSimpIndexProc$, and $CParamProc\ (N(Exp^{+}))$ are not yet implemented.

$$\Upsilon_{CProc}(P1 \Box P2) \mathrel{\widehat=} \Upsilon_{CProc}(P1)\ \texttt{[]}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(P1 \setminus CS) \mathrel{\widehat=} \Upsilon_{CProc}(P1)\ \texttt{\textbackslash}\ \Upsilon_{Pred_{CS}}(CS)$$
$$\Upsilon_{CProc}(P1 \sqcap P2) \mathrel{\widehat=} \Upsilon_{CProc}(P1)\ \texttt{|\textasciitilde|}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(P1 \interleave P2) \mathrel{\widehat=} \Upsilon_{CProc}(P1)\ \texttt{|||}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(P) \mathrel{\widehat=} \texttt{P}$$
$$\Upsilon_{CProc}(P1 \llbracket CS \rrbracket P2) \mathrel{\widehat=} \Upsilon_{CProc}(P1)\ \texttt{[|}\ \Upsilon_{Pred_{CS}}(CS)\ \texttt{|]}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(P1\ ;\ P2) \mathrel{\widehat=} \Upsilon_{CProc}(P1)\ \texttt{;}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(\Box\, x : S \bullet P1) \mathrel{\widehat=} \texttt{[]}\ \ \texttt{x}\ \ \texttt{:}\ \Upsilon_{ZExpr}(S)\ \texttt{@}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(\sqcap\, x : S \bullet P1) \mathrel{\widehat=} \texttt{|\textasciitilde|}\ \ \texttt{x}\ \ \texttt{:}\ \Upsilon_{ZExpr}(S)\ \texttt{@}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(\left|\left|\left| x : S \bullet P1\right.\right.\right.) \mathrel{\widehat=} \texttt{|\textasciitilde|}\ \ \texttt{x}\ \ \texttt{:}\ \Upsilon_{ZExpr}(S)\ \texttt{@}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(\llbracket CS \rrbracket\, x : S \bullet P1) \mathrel{\widehat=} \texttt{[|}\ \Upsilon_{Pred_{CS}}(CS)\ \texttt{|]}\ \ \texttt{x}\ \ \texttt{:}\ \Upsilon_{ZExpr}(S)\ \texttt{@}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(;\ x : S \bullet P1) \mathrel{\widehat=} \texttt{;}\ \ \texttt{x}\ \ \texttt{:}\ \Upsilon_{ZExpr}(S)\ \texttt{@}\ \Upsilon_{CProc}(P2)$$
$$\Upsilon_{CProc}(\mathbf{begin}\ AL \bullet MA\ \mathbf{end}) \mathrel{\widehat=}$$
$$\quad \texttt{let}\ \Upsilon_{PPar_list}(AL)$$
$$\quad \texttt{within}\ \Upsilon_{CAction}(MA)$$
$$\Upsilon_{CProc}(\mathbf{begin} \bullet MA\ \mathbf{end}) \mathrel{\widehat=} \Upsilon_{CAction}(MA)$$
$$\Upsilon_{CProc}(Proc[NL := EL]) \mathrel{\widehat=} \texttt{P[[}\ \Upsilon_{Rename}(NL, EL)\ \texttt{]]}$$

provided

- P is a process name.
- $P1$ and $P2$ is a process $CProc$
- cs is a channel set $CSExp$
- MA is the main action of the *Circus* process.
- AL is the list of actions.

This function maps any renaming, to its equivalent syntax in CSP_M.

$$\Upsilon_{Rename}(x \mathbin{\#} xs, y \mathbin{\#} xs) \mathrel{\widehat=} \Upsilon_{Comm}(x)\ \texttt{<-}\ \ \Upsilon_{Comm}(y)\texttt{,}\ \ \Upsilon_{Rename}(xs, xs)$$
$$\Upsilon_{Rename}([x], [y] =) \mathrel{\widehat=} \Upsilon_{Comm}(x)\ \texttt{<-}\ \ \Upsilon_{Comm}(y)$$

C.1.6 Mapping *Circus* Processes Paragraphs

$$\Upsilon_{PPar}(P \mathrel{\widehat=} Decl \bullet A) \mathrel{\widehat=} \texttt{P(}\Upsilon_{ZGenFilt_list}(Decl)\texttt{)}\ \ \texttt{=}\ \ \Upsilon_{CAction}(A)$$
$$\Upsilon_{PPar}(P \mathrel{\widehat=} A) \mathrel{\widehat=} \texttt{P}\ \ \texttt{=}\ \ \Upsilon_{CAction}(A)$$

C.1.7 Mapping *Circus* Actions

$$
\begin{array}{rcl}
\Upsilon_A(c \rightarrow\ A) & \widehat{=} & \texttt{c -> } \Upsilon_A(A) \\
\Upsilon_A(c?x \rightarrow A) & \widehat{=} & \texttt{c?x -> } \Upsilon_A(A) \\
\Upsilon_A(c.v \rightarrow A) & \widehat{=} & \texttt{c.v -> } \Upsilon_A(A) \\
\Upsilon_A(c?x : P \rightarrow A) & \widehat{=} & \texttt{c?x}:\Upsilon_{ZExp}(P)\texttt{->}\ \Upsilon_A(A) \\
\Upsilon_A(mget.x?v_x : \delta(x) \rightarrow A) & \widehat{=} & \texttt{mget.x?v_x:(typeTYP(x)) -> } \Upsilon_A(A) \\
\Upsilon_A(mset.x.v_x \rightarrow A) & \widehat{=} & \texttt{mset.x.(TYP.valueTYP(v_x)) -> } \Upsilon_A(A) \\
\Upsilon_A(A \Box B) & \widehat{=} & \Upsilon_A(A)\,\texttt{[]}\,\Upsilon_A(B) \\
\Upsilon_A(A \sqcap B) & \widehat{=} & \Upsilon_A(A)\,\texttt{|\~{}|}\,\Upsilon_A(B) \\
\Upsilon_A(A \setminus cs) & \widehat{=} & \Upsilon_A(A)\setminus\Upsilon_{\mathbb{P}cs}(cs) \\
\Upsilon_A(g\ \&\ A) & \widehat{=} & \Upsilon_{\mathbb{B}}(g)\ \texttt{\&}\ \ \Upsilon_A(A) \\
\Upsilon_A(A \parallel\!\![\ ns1 \mid ns2\ ]\!\!\parallel B) & \widehat{=} & \Upsilon_A(A)\,\texttt{|||}\,\Upsilon_A(B) \\
\Upsilon_A(A \llbracket\ ns1 \mid cs \mid ns2\ \rrbracket B) & \widehat{=} & \Upsilon_A(A)\,\texttt{[|}\,\Upsilon_{\mathbb{P}cs}(cs)\,\texttt{|]}\,\Upsilon_A(B) \\
\Upsilon(\mu X \bullet\ A) & \widehat{=} & \texttt{let mu}A\texttt{=}\Upsilon_A(A)\,\texttt{within mu}A \\
\Upsilon_A(\Box x : S \bullet A) & \widehat{=} & \texttt{[] x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A) \\
\Upsilon_A(\sqcap x : S \bullet A) & \widehat{=} & \texttt{|\~{}| x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A) \\
\Upsilon_A(\left|\left|\left| x : S \bullet \llbracket \varnothing \rrbracket A\right.\right.\right.) & \widehat{=} & \texttt{||| x:}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A) \\
\Upsilon_A(\llbracket cs \rrbracket\ x : S \bullet A) & \widehat{=} & \texttt{[|}\Upsilon_{\mathbb{P}cs}(cs)\texttt{|] x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A) \\
\Upsilon_A(\llbracket cs \rrbracket\ x : S \bullet \llbracket \varnothing \rrbracket A) & \widehat{=} & \texttt{[|}\Upsilon_{\mathbb{P}cs}(cs)\texttt{|] x :}\Upsilon_{\mathbb{P}}(S)\,\texttt{@}\,\Upsilon_A(A) \\
\Upsilon_A(\text{;}\ \ x : S \bullet A) & \widehat{=} & \texttt{; x :}\Upsilon_{seq}(S)\,\texttt{@}\,\Upsilon_A(A) \\
\Upsilon_A(A\ ;\ B) & \widehat{=} & \Upsilon_A(A)\,\texttt{;}\,\Upsilon_A(B) \\
\Upsilon_A(\text{Skip}) & \widehat{=} & \texttt{SKIP} \\
\Upsilon_A(\text{Stop}) & \widehat{=} & \texttt{STOP} \\
\Upsilon_A(\text{Chaos}) & \widehat{=} & \texttt{CHAOS}
\end{array}
$$

C.1.8 Mapping *Circus* Commands

Assignments are translated into prefixed actions with *mget* and *mset* with the Ω functions. Therefore, we only have to translate conditional commands with *if* $-$ *then* $-$ *else*. Moreover, we do not have to provide any translation for local variable declaration **var** as they are promoted to the *Memory* state process.

$$
\Upsilon_A\left(\begin{array}{l}\textbf{if } g_0 \longrightarrow A_0 \\ \quad [\!]\ldots \\ \quad [\!] g_n \longrightarrow A_n \\ \textbf{fi}\end{array}\right) \widehat{=} \left(\begin{array}{l} g_0\ \texttt{->}\ \Upsilon_A(A_0) \\ \texttt{[]}\ \ldots \\ \texttt{[]}\ g_n\ \texttt{->}\ \Upsilon_A(A_n)\end{array}\right)
$$

C.1.9 Mapping *Circus* Guarded Actions

$$
\Upsilon_A\left(\begin{array}{l}(g_0)\ \&\ A_0 \\ [\!]\ldots \\ [\!](g_0)\ \&\ A_n\end{array}\right) \widehat{=} \left(\begin{array}{l} g_0\ \ \texttt{\&}\ \ \ \Upsilon_A(A_0) \\ \texttt{[]}\ \ldots \\ \texttt{[]}\ g_n\ \ \texttt{\&}\ \ \ \Upsilon_A(A_n)\end{array}\right)
$$

C.2 Mapping Functions from *Circus* to CSP - Based on D24.1 - COMPASS

The definitions mapping functions for Expressions and the ones for Predicates were expanded in order to cover a wider range of constructs. These were based on the Z definitons and some may use the auxiliary definitons in CSP_M included in the `functions_aux.csp` and `sequences_aux.csp`.

C.2.1 Mapping Function for Expressions

Expression		Mapping	Condition
$\Upsilon_{ZExpr}(\mathbb{N})$	$\widehat{=}$	`NatValue`	
$\Upsilon_{ZExpr}(m)$	$\widehat{=}$	m	$m \in \mathbb{Z}$
$\Upsilon_{ZExpr}(a_T)$	$\widehat{=}$	`value(`$\mathit{def_U_prefix}\ T$`)(v_`a`)`	$a_T \in \mathit{State} \wedge T \in \mathit{UNIVERSE}$
$\Upsilon_{ZExpr}(a_t)$	$\widehat{=}$	a	$a \notin \mathit{State}$
$\Upsilon_{ZExpr}(m * n)$	$\widehat{=}$	`(`$\Upsilon_{ZExpr}(n)$`*` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(m + n)$	$\widehat{=}$	`(`$\Upsilon_{ZExpr}(n)$`+` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(m - n)$	$\widehat{=}$	`(`$\Upsilon_{ZExpr}(n)$`-` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(a \mapsto b)$	$\widehat{=}$	`(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(\# a)$	$\widehat{=}$	`card(`$\Upsilon_{ZExpr}\, a$`)`	
$\Upsilon_{ZExpr}(a \bigcap b)$	$\widehat{=}$	`Inter(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(a \bigcup b)$	$\widehat{=}$	`Union(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(a \frown b)$	$\widehat{=}$	$\Upsilon_{ZExpr}(a)$`^`$\Upsilon_{ZExpr}(b)$	
$\Upsilon_{ZExpr}(a \cap b)$	$\widehat{=}$	`inter(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(a \cup b)$	$\widehat{=}$	`union(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(\frown / s)$	$\widehat{=}$	`concat(`$\Upsilon_{ZExpr}(s)$`)`	
$\Upsilon_{ZExpr}(m \operatorname{div} n)$	$\widehat{=}$	`(`$\Upsilon_{ZExpr}(n)$`/` $\Upsilon_{ZExpr}(m$`)`	
$\Upsilon_{ZExpr}(\operatorname{dom} a)$	$\widehat{=}$	`dom(`$\Upsilon_{ZExpr}(a)$`)`	
$\Upsilon_{ZExpr}(\operatorname{ran} a)$	$\widehat{=}$	`ran(`$\Upsilon_{ZExpr}\, a$`)`	
$\Upsilon_{ZExpr}(m \bmod n)$	$\widehat{=}$	$\Upsilon_{ZExpr}(n)$`%` $\Upsilon_{ZExpr}(m)$	
$\Upsilon_{ZExpr}(m \setminus n)$	$\widehat{=}$	`diff(`$\Upsilon_{ZExpr}(n)$`,` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(m \lhd n)$	$\widehat{=}$	`dres(`$\Upsilon_{ZExpr}(n)$`,` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(m \mathbin{;} n)$	$\widehat{=}$	`comp(`$\Upsilon_{ZExpr}(n)$`,` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(m \rhd n)$	$\widehat{=}$	`rres(`$\Upsilon_{ZExpr}(n)$`,` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(m \pfun n)$	$\widehat{=}$	`pfun(`$\Upsilon_{ZExpr}(n)$`,` $\Upsilon_{ZExpr}(m)$`)`	
$\Upsilon_{ZExpr}(- n)$	$\widehat{=}$	`-`$(\Upsilon_{ZExpr}(n))$	
$\Upsilon_{ZExpr}(a \oplus \{b \mapsto c\})$	$\widehat{=}$	`over(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`,`$\Upsilon_{ZExpr}(c)$`)`	
$\Upsilon_{ZExpr}(a \times b)$	$\widehat{=}$	`(`$\Upsilon_{ZExpr}(a)$`.`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(a \oplus b)$	$\widehat{=}$	`oplus(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(\mathbb{P}\, a)$	$\widehat{=}$	`Set(`$\Upsilon_{ZExpr}\, a$`)`	
$\Upsilon_{ZExpr}(\mathbf{seq}\, a)$	$\widehat{=}$	`Seq(`$\Upsilon_{ZExpr}\, a$`)`	
$\Upsilon_{ZExpr}(a \setminus b)$	$\widehat{=}$	`diff(`$\Upsilon_{ZExpr}(a)$`,`$\Upsilon_{ZExpr}(b)$`)`	
$\Upsilon_{ZExpr}(a \mathinner{..} b)$	$\widehat{=}$	`{`$\Upsilon_{ZExpr}(a)$`..`$\Upsilon_{ZExpr}\, b$`}`	
$\Upsilon_{ZExpr}(\varnothing)$	$\widehat{=}$	`{}`	
$\Upsilon_{ZExpr}(\{ a, \ldots, c \})$	$\widehat{=}$	`{` $a, \ldots, c$ `}`	
$\Upsilon_{ZExpr}(\langle\rangle)$	$\widehat{=}$	`<>`	
$\Upsilon_{ZExpr}(\langle b_a \rangle)$	$\widehat{=}$	`<`b_a`>`	$b_a \in \mathit{BINDINGS}$
$\Upsilon_{ZExpr}(\langle sv_a \rangle)$	$\widehat{=}$	`<`sv_a`>`	$sv_a \in \mathit{NAMES}$
$\Upsilon_{ZExpr}(\langle ns \rangle)$	$\widehat{=}$	`<y | y <- ns>`	$ns \in \mathit{NSExp}$
$\Upsilon_{ZExpr}(\mathbf{seq}\, A \frown \mathbf{seq}\, B)$	$\widehat{=}$	$\Upsilon_{ZExpr}(A)$`^` $\Upsilon_{ZExpr}(B)$	
$\Upsilon_{ZExpr}(\mathbf{seq}\, a)$	$\widehat{=}$	`<y | y <-` a`>`	
$\Upsilon_{ZExpr}(\langle a, \ldots, c \rangle)$	$\widehat{=}$	`<`$a, \ldots, c$`>`	
$\Upsilon_{ZExpr}(A \times B)$	$\widehat{=}$	A`.`B	
$\Upsilon_{ZExpr}((a, \ldots, c))$	$\widehat{=}$	`(`$a, \ldots, c$`)`	
$\Upsilon_{ZExpr}(fa)$	$\widehat{=}$	$\Upsilon_{ZExpr}(f)$`(`$\Upsilon_{ZExpr}(a)$`)`	

C.2.2 Mapping Functions for Predicates

$$
\begin{aligned}
\Upsilon_{ZPred}(\textbf{if } x \textbf{ then } y \textbf{ else } y) &\mathrel{\widehat=} \texttt{if } \Upsilon_{ZPred}(x)\texttt{then}\Upsilon_{ZPred}(y)\texttt{else}\Upsilon_{ZPred}(z)\\
\Upsilon_{ZPred}(a \geq b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{>=}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a > b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{>}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a \leq b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{<=}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a < b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{<}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a \neq b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{!=}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a \geq b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{>=}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a = b) &\mathrel{\widehat=} \Upsilon_{ZExpr}(a)\texttt{==}\Upsilon_{ZExpr}(b)\\
\Upsilon_{ZPred}(\neg\, a) &\mathrel{\widehat=} \texttt{not (}\Upsilon_{ZPred}(a)\texttt{)}\\
\Upsilon_{ZPred}(a \wedge b) &\mathrel{\widehat=} \Upsilon_{ZPred}(a)\texttt{and }\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(a \vee b) &\mathrel{\widehat=} \Upsilon_{ZPred}(a)\texttt{or}\Upsilon_{ZPred}(b)\\
\Upsilon_{ZPred}(\textbf{True}) &\mathrel{\widehat=} \texttt{true}\\
\Upsilon_{ZPred}(\textbf{False}) &\mathrel{\widehat=} \texttt{true}\\
\Upsilon_{ZPred}(a \in B) &\mathrel{\widehat=} \texttt{member(}\Upsilon_{ZExpr}(a)\texttt{,}\Upsilon_{ZExpr}(B)\texttt{)}\\
\Upsilon_{ZPred}(a) &\mathrel{\widehat=} a
\end{aligned}
$$

Appendix D

Circus refinement laws

We present a list of *Circus* Refinement Laws used throughout this thesis. With the exception of Law 20 and Law 21, which are part of our contribution, the other laws may be found in the Deliverable 24.1 [132].

Law 1 (Parallelism composition/External choice—expansion*)

$$(\Box\, i \bullet a_i \rightarrow A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\Box\, j \bullet b_j \rightarrow B_j)$$
$$=$$
$$(\Box\, i \bullet a_i \rightarrow A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket ((\Box\, j \bullet b_j \rightarrow B_j) \Box (c \rightarrow C))$$

provided

- $\bigcup_i \{a_i\} \subseteq cs$
- $c \in cs$
- $c \notin \bigcup_i \{a_i\}$
- $c \notin \bigcup_j \{b_j\}$

Law 2 (Channel extension 3*)

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2(e)) \setminus cs_2$$
$$=$$
$$((c!e \rightarrow A_1) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (c?x \rightarrow A_2(x))) \setminus cs_2$$

provided

- $c \in cs_1$
- $c \in cs_2$
- $x \notin FV(A_2)$

Law 3 (Sequence unit)

$$(A) Skip;\ A$$
$$(B) A = A;\ Skip$$

Law 4 (True guard)

$$(true) \,\&\, A = A$$

Law 5 (External choice/Sequence—distribution)

$$(\Box\, i \bullet \big(g_i\big) \,\&\, c_i \rightarrow A_i);\ B = \Box\, i \bullet \big(g_i\big) \,\&\, c_i \rightarrow A_i;\ B$$

Law 6 (Parallelism composition/External choice—distribution*)

$$\Box\, i \bullet (A_i \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A) = (\Box\, i \bullet A_i) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A$$

provided

- $initials(A) \subseteq cs$
- A is deterministic

Law 7 (Hiding/External choice—distribution*)

$$(A_1 \Box A_2) \setminus cs = (A_1 \setminus cs) \Box (A_2 \setminus cs)$$

provided

- $(initials(A_1) \cup initials(A_2)) \cap cs = \varnothing$

Law 8 (Hiding/Sequence—distribution*)

$$(A_1\ ;\ \ A_2) \setminus cs = (A_1 \setminus cs)\ ;\ \ (A_2 \setminus cs)$$

Law 9 (Hiding Identity*)

$$A \setminus cs = A$$

provided

- $cs \cap usedC(A) = \varnothing$

Law 10 (Parallelism composition/Sequence—step*)

$$(A_1;\ A_2) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_3 = A_1;\ (A_2 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_3)$$

provided

- $initials(A_3) \subseteq cs$
- $cs \cap usedC(A_1) = \varnothing$
- $wrtV(A_1) \cap usedV(A_3) = \varnothing$
- A_3 is divergence-free
- $wrtV(A_1) \subseteq ns_1$

Law 11 (Variable Substitution2*)

$$\mathbf{var}\ x \bullet A(x) = \mathbf{var}\ y \bullet A(y)$$

provided

- x is not free in A
- y is not free in A

Law 12 (Assignment Removal*)

$x := e \,;\; A(x) = A(e)$

provided x is not free in $A(e)$

Law 13 (Innocuous Assignment*)

$x := x = \text{Skip}$

Law 14 (Variable block introduction*)

$\textbf{var}\;\; x : T \bullet A = AA = \textbf{var}\;\; x : T \bullet A$

provided $x \notin FV(A)$

Law 15 (Guard combination) 1

$(g_1) \mathbin{\&} ((g_2) \mathbin{\&} A) = ((g_1 \wedge g_2)) \mathbin{\&} A$

Law 16 (Variable block/Sequence—extension*) 1

$A_1 \,;\; (\textbf{var}\;\; x : T \bullet A_2) \,;\; A_3 = (\textbf{var}\;\; x : T \bullet A_1 \,;\; A_2 \,;\; A_3)$

provided $x \notin FV(A_1) \cup FV(A_3)$

Law 17 (prom-var-state) [0]

$$\begin{array}{l} \textbf{begin} \\ \quad (\textbf{state}\, S) \\ \quad L(x : T) \\ \quad \bullet\, (\textbf{var}\, x : T \bullet MA) \\ \textbf{end} \end{array} = \begin{array}{l} \textbf{begin} \\ \quad (\textbf{state}\, S \wedge [x : T]) \\ \quad L(_) \\ \;\bullet\, MA \\ \textbf{end} \end{array}$$

Law 18 (prom-var-state-2) [0]

$$\begin{array}{l} \textbf{begin} \\ \quad L(x : T) \\ \quad \bullet\, (\textbf{var}\, x : T \bullet MA) \\ \textbf{end} \end{array} = \begin{array}{l} \textbf{begin} \\ \quad (\textbf{state}[x : T]) \\ \quad L(_) \\ \quad \bullet\, MA \\ \textbf{end} \end{array}$$

Law 19 (Guard/Parallelism composition—distribution*) 3

$((g) \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = (g) \mathbin{\&} (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$

provided $initials(A2) \subseteq cs$

Law 20 (Process Splitting 3 $ArturGomes$**)**

provided x is a local variable, and therefore, can be promoted into a parameter of the process P

$$\begin{array}{l} \textbf{process}\;\; G \mathrel{\widehat=} \textbf{begin}\, PPar \bullet \textbf{var}\, x : T \bullet P(x)\, \textbf{end} \\ = \\ \textbf{process}\;\; G \mathrel{\widehat=} \textbf{begin}\, PPar \bullet \sqcap\, x : T \bullet P(x)\, \textbf{end} \\ = \\ \textbf{process}\;\; G \mathrel{\widehat=}\; \sqcap\, x : T \bullet G'(x) \\ \textbf{process}\;\; G' \mathrel{\widehat=}\; (x : T \bullet \textbf{begin}\, PPar \bullet P(x)\, \textbf{end}) \end{array}$$

Law 21 (Process Splitting 4 *ArturGomes***)**

$$\textbf{process}\ \ G \mathrel{\widehat=} \left(\begin{array}{l} \textbf{begin} \\ \quad PPar \\ \quad \bullet\ \textbf{var}\ x : T \bullet P(x) \\ \textbf{end} \end{array} \right)$$
$$=$$
$$\textbf{process}\ \ G \mathrel{\widehat=}\ x : T \bullet \left(\begin{array}{l} \textbf{begin} \\ \quad PPar \\ \quad \bullet\ P(x) \\ \textbf{end} \end{array} \right)$$

D.1 Lemmas from Deliverable 24.1 [132]

Lemma (K.2)

$$\left(\begin{array}{l} A \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\ Memory(b) \end{array} \right)$$
$$=$$
$$\left(\begin{array}{l} A \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\ \left(\begin{array}{l} (\Box\, n : NAME \bullet mget.n!b(n) \rightarrow Memory(b)) \\ \Box(\Box\, n : NAME \bullet mset.n?nv \rightarrow Memory(b \oplus \{n \mapsto nv\})) \\ \Box terminate \rightarrow \mathrm{Skip} \end{array} \right) \end{array} \right) \setminus MEM_I$$

D.2 *Circus* Denotational Semantics [129, Chapter 3]

Definition 1 (B.41)

$$\textbf{begin state}\,[decl \mid pred]\,PPars\ \bullet\ A\ \textbf{end} \mathrel{\widehat=} \textbf{var}\ decl \bullet A$$

Appendix E

Beyond the Tool Implementation - Linking with Isabelle/UTP

Appendix F

From Haskell to Isabelle using Haskabelle

Because we are using Haskell, we could use Isabelle/HOL as our target theorem prover. We can use one of its tools, Haskabelle [79], which is capable of translating Haskell code into the ML-like syntax of Isabelle, which is similar to Haskell. Moreover, another benefit from using Isabelle is the possibility of using Isabelle/UTP, which is an implementation of the Unifying Theories of Programming (UTP) [83], as well as the theories of *Circus*.

The result from Haskabelle is a very similar code compared to the original Haskell implementation. Then, we would load the theories for Isabelle/UTP and create a function that maps our Haskell syntax of *Circus* into that used by Isabelle/UTP.

Unfortunately, Haskabelle is no longer being maintained by its developers, and the last release was delivered as a package of Isabelle2015. Later versions of IsabelleHOL no longer contain the tool packages. However, one can still execute Haskabelle on Isabelle2015 and load the translated files on later versions of IsabelleHOL.

Although, as the Haskabelle code is open-source, we updated its files to comply with GHC 7.10, and manually (and informally) inserted Haskabelle into Isabelle2017. Such a fix required the manual installation of older versions of a few Haskell packages. We have a dedicated folder within our repository for the fixed Haskabelle code, which we plan to submit to the Isabelle/HOL developers for a possible reinclusion in its future releases.

Our first task was to resolve any Haskell library dependency, as the *Haskabelle* tool does not support importing functions from Haskell libraries. Instead, we have to make a local copy of the required librares. One of the most used is the *List* library, and thus, we organise all the list-related functions in a local auxiliary file.

One of the caveats in the Haskabelle tool is that some Haskell constructs are not directly translated. For instance, an equality in a case expression, **case** $(a = b)$ **of**... is translated as **case** $(eq\ a\ b)$ **of**... and as it is, the definition of function eq, from the Haskabelle's *Prelude.thy* file clashes with the types of a and b from our code. We need to manually replace the $eq\ a\ b$ by $a = b$, so that the code is accepted by Isabelle.

Another caveat found is the translation of tuples into Isabelle's notation. The native translation

with Haskabelle transforms, for example, the tuple $(T1 * T2 * T3)$ into $((T1 * T2) * T3)$, whose function definition in Isabelle differs from the one defined in our implementation in Haskell, and therefore, the native translation is not equivalent. Thus, we manually fix any definition of tuples in our code, so the function behaves as the expected ones written in Haskell. The resulted code has a similar structure compared to the original Haskell code, preserving the dependency tree of the local libraries. The Haskell implementation of the three Ω functions along with the example of the Ω' function presented in the section above is defined in the code extract below.

F.1 Reasoning using Isabelle/UTP

Our tool, *Circus2CSP*, was built based on the implementations of formal rules for remodelling *Circus*, along with the *Circus* refinement calculus. All those rules and laws were proved to be correct, some of them by hand, some with the help of theorem provers [176, 130]. However, we know that theory and practice do not always result in the same product. Therefore, we would like to verify if the implementation in Haskell of all those theories used is correct *w.r.t.* its formal definitions from the literature.

Moreover, our refinement calculator is not able to discharge the proof obligations derived from the refinement laws applied in its calculations. We, however, developed our tool in such a way that those proof obligations would be made available in a file so that they can be, later on, parsed an prepared as lemmas for external use in a theorem prover.

Therefore, we can list three goals we would like to achieve while using Isabelle/UTP :

1. Verifying the correctness of Haskell implementation of *Circus2CSP*
2. Discharging the proof obligations for the automatic *Circus* refinement calculator
3. Discharging any proof obligation while refining Z schemas into *Circus*

We first introduce here briefly how we translated into Isabelle/HOL our Haskell AST of *Circus* as well as the omega functions and the code for the refinement calculator. We illustrate the similarities in both sides of the translation from Haskell to Isabelle with a small piece of code from the omega functions. As an example, we first show a few selected Haskell functions of the omega transformations.

```
omega_CAction :: CAction -> CAction
omega_CAction CSPSkip = CSPSkip
omega_CAction (CSPCommAction (ChanComm c []) a)
  = (CSPCommAction (ChanComm c []) (omega_CAction a))
omega_CAction (CSPExtChoice ca cb)
  = make_get_com lsx (CSPExtChoice (omega_prime_CAction ca)
                                   (omega_prime_CAction cb))
    where
    lsx=remdups.concat.map get_ZVar_st (free_var_CAction (CSPExtChoice ca cb))
omega_CAction (CSPHide a cs) = (CSPHide (omega_CAction a) cs)
omega_CAction x = x
omega_prime_CAction :: CAction -> CAction
omega_prime_CAction (CSPSeq ca cb)
  = (CSPSeq (omega_prime_CAction ca) (omega_CAction cb))
omega_prime_CAction x = omega_CAction x
```

The above Haskell code is translated into Isabelle/UTP via Haskabelle, resulting in the equivalent functions *omega_CAction* and *omega_prime_CAction* in the syntax of Isabelle/HOL. Both functions are mutually recursive as the *CSPSeq* function calls both *omega_CAction* and *omega_prime_CAction*.

```
function (sequential) omega_CAction :: "CAction ⇒ CAction" and
    omega_prime_CAction :: "CAction ⇒ CAction"
where
  "omega_CAction CSPSkip = CSPSkip"
| "omega_CAction (CSPCommAction (ChanComm c Nil) a)
   = (CSPCommAction (ChanComm c Nil) (omega_CAction a))"
| "omega_CAction (CSPExtChoice ca cb)
   = make_get_com lsx (CSPExtChoice (omega_prime_CAction ca)
                                    (omega_prime_CAction cb))
  where
   lsx=remdups.concat.map get_ZVar_st (free_var_CAction(CSPExtChoice ca cb))"
| "omega_CAction (CSPHide a cs) = (CSPHide (omega_CAction a) cs)"
| "omega_CAction x = x"
| "omega_prime_CAction (CSPSeq ca cb)
   = (CSPSeq (omega_prime_CAction ca) (omega_CAction cb))"
| "omega_prime_CAction x = omega_CAction x"
by pat_completeness auto
termination by size_change
```

Within the environment of Isabelle/UTP, we developed a function for mapping the specifications written in the syntax of *Circus2CSP* into the one used for writing *Circus* in Isabelle/UTP. Such a function, herein *toUTP*, was prototyped with only a few of the *Circus* actions constructs in order to experiment the overall approach for reaching Isabelle and therefore, for formally verifying *Circus2CSP*. The *toUTP* function, for now, is limited to the following operators:

1. *Skip*, *Chaos* and *Stop*
2. Internal choice, external choice and sequential composition
3. Indexed operators for interleaving and internal choice
4. Simple prefix events

The structure of *toUTP* is illustrated in Fig F.1. We defined on purpose the clause for external choice with a symmetrical equivalence. We expected that Isabelle/UTP would use the *Circus* refinement laws and in fact, it does prove that such symmetry is valid.

```
fun toUTP :: "CAction ⇒ (ZName, ZName) action"
  where
  "toUTP CSPSkip = Skip"
  | "toUTP CSPChaos = Chaos"
  | "toUTP CSPStop = Stop"
  | "toUTP (CSPExtChoice a b) =  (toUTP b) □ (toUTP a)"
  | "toUTP (CSPSeq a b) =  (toUTP a) ;; (toUTP b)"
  | "toUTP (CSPIntChoice a b) =  (toUTP a) ⊓ (toUTP b)"
  | "toUTP (CSPRepSeq xs p) =  (;;c x : xs • (toUTP p))"
  | "toUTP (CSPRepInterl xs p) =  (||| x : xs • (toUTP p))"
  | "toUTP (CSPCommAction (ChanComm n Nil) d) = PrefixCSP «n» (toUTP d)"
```

Figure F.1: Mapping Haskell to Isabelle/UTP

Because we decided to prototype *toUTP* with simple constructs, the proofs for small examples of *Circus* actions using those constructs listed were quite trivial. For example, the fact that we introduced a symmetry in the external choice operator in *toUTP* was not a problem for the prover, since such property was already defined in Isabelle, and was proved using simplification tactics.

We illustrate the bridge between the functions implemented in *Circus2CSP* and the theories of Isabelle/UTP with the example of the action *Skip* ; *Skip*, which can be refined into *Skip*. We proved that,

when applying *toUTP* to the Haskell description of that action, (`CSPSeq CSPSkip CSPSkip`), it can be refined into *Skip* in the syntax of Isabelle/UTP.

```
lemma "toUTP (CSPSeq CSPSkip CSPSkip) ⊑ Skip"
  apply simp
  apply (metis CSP3_Skip CSP3_def Healthy_if eq_refl)
done
```

The proof first transforms the left-hand side into the syntax of Isabelle, and then, using the built-in theories of *Circus*, it proves the refinement.

Now that we presented our work on linking our Haskell implementation of the translation into Isabelle, we now focus on the overall strategy for verifying that implementation using Isabelle/UTP.

F.1.1 Considerations for Future Work

From that point, our next task would be to introduce the omega mapping functions into the Isabelle/UTP environment, along with any *Circus* refinement law used in our automatic refinement calculator that would not be yet available in Isabelle. Finally, with those steps concluded, we would be able to start with the proofs for supporting our verified translator, as illustrated in Figure 8.1.

As we mentioned earlier, we would benefit from Isabelle while discharging the proofs related to the refinement of both *Circus* and Z schemas using ZRC. Once able to verify the correctness of our tool, we could start implementing a module of *Circus2CSP* capable of verifying if the laws applied in the refinement steps are valid. For such, we can extract every refinement step and its provisos, which are the required conditions to prove that step is true, and format them into the syntax of a lemma in Isabelle. Therefore, we may have at the end, a list of lemmas one for each refinement step to be proved using Isabelle. Hopefully, we will be able to prove that, given $P \sqsubseteq Q$, P is refined by Q by rewriting them using the intermediate steps proved as lemmas.

We illustrate below an example of one of the *Circus* refinement laws, initially implemented in Haskell, then imported to Isabelle. First we introduce how the *Refinement* type is defined in Isabelle - we had to change the names of *Done* and *None* to *Done* and *NoRef*, as *None* is a reserved word in the context of Isabelle. Therefore, we use *'t Refinement* for a refinement of either a *CAction* or a *CProcess*.

```
datatype ('t) Refinement
  = NoRef
  | Done "'t option" "'t option" "ZPred list"
```

Then, we illustrate the implementation of the law `sequence-unit`, L. 3. It starts with a *Circus* action in the shape of (`CSPSeq CSPSkip act`) or (`CSPSeq act CSPSkip`), for **Skip** ; *Act* and *Act* ; **Skip** respectively, and when applied, it results in a *CAction Refinement*, with only *Act*.

```
 (* Implementation of Skip ; Act *)
 fun crl_seqSkipUnit_a :: "CAction ⇒ CAction Refinement"
 where
   "crl_seqSkipUnit_a (CSPSeq CSPSkip a) = Done (Some (CSPSeq CSPSkip a)) (Some a) Nil "
 | "crl_seqSkipUnit_a _ = NoRef"

(* Implementation of Act ; Skip *)
 fun crl_seqSkipUnit_b :: "CAction ⇒ CAction Refinement"
 where
    "crl_seqSkipUnit_b (CSPSeq a CSPSkip) = Done (Some (CSPSeq a CSPSkip)) (Some a) Nil"
   | "crl_seqSkipUnit_b _ = NoRef"
```

And the refinement of, for example, Skip ; *terminate* $\rightarrow$ Skip, when applying `crl_seqSkipUnit_a` is, therefore, *terminate* $\rightarrow$ Skip, as illustrated in Figure F.2.

```
1  value "crl_seqSkipUnit_a
     (CSPSeq
3      CSPSkip
       (CSPCommAction (ChanComm ''terminate'' []) CSPSkip))"
```

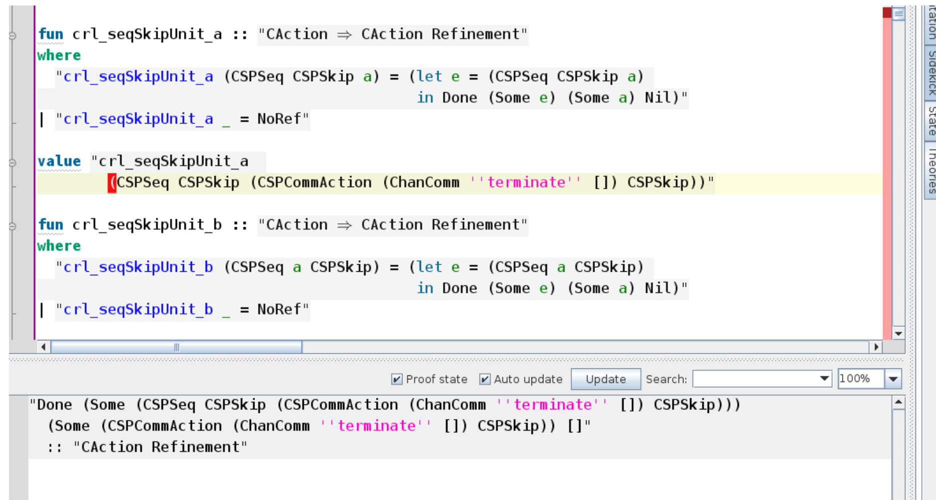


Figure F.2: Refinement test

We also illustrate how the Haskell implementation of the refinement calculator, translated to the environment of Isabelle, can be proved to be correct using the refinement laws of *Circus*in Isabelle. We use the above presented equation for applying the law `sequence-unit`, as illustrated in Figure F.2, and prove its equivalence to the syntax of Isabelle/UTP. The action $PrefixCSP \ll'' terminate'' \gg Skip$ is a syntactic sugar for $terminate \rightarrow Skip$. And as expected the proof is concluded using the *simp* command of Isabelle/UTP.

```
   lemma "toUTP (fromDone(refined (crl_seqSkipUnit_a
2                    (CSPSeq CSPSkip (CSPCommAction (ChanComm ''terminate'' []) CSPSkip)))))
           = PrefixCSP ≪''terminate''≫ Skip"
4    by simp
```

Appendix G

Proofs on *Circus* models

G.1 Proofs on the Z schemas refinement - *Chronometer*

Here we present the proofs of the refinement of Z schemas for the *Chronometer* process as discussed in Section 3.5.1.

Init action

$Init$

$=$ [Init schema def]

$[\Delta State \mid sec' = 0 \land min' = 0]$

$=$ [basic conversion bC [34]]

$sec, min : [(inv \land \exists sec', min' \bullet sec' = 0 \land min' = 0 \land inv'), (inv' \land sec' = 0 \land min' = 0)]$

$=$ [predicate calculus]

$sec, min : [(\exists sec', min' \bullet sec' = 0 \land min' = 0), (sec' = 0 \land min' = 0)]$

$=$ [Assignment conversion $assC$ [34], provided $inv[0, 0/min, sec]$]

$(sec, min := 0, 0)$

IncSec action

$IncSec$

$=$ [Init schema def]

$[\Delta State \mid sec' = (sec + 1)]$

$=$ [basic conversion bC [34]]

$sec, min : [(inv \land \exists sec', min' \bullet sec' = (sec + 1) \land inv'), (inv' \land sec' = (sec + 1))]$

$=$ [L 5.4 - contract frame [119]]

$sec : [(inv \land \exists sec', min' \bullet sec' = (sec + 1) \land inv')[min/min'], (inv' \land sec' = (sec + 1))[min/min']]$

$=$ [predicate calculus]

$sec : [(\exists sec' \bullet sec' = (sec + 1)), (sec' = (sec + 1))]$

$=$ [Assignment conversion $assC$ [34], provided $inv[(sec + 1)/sec]$]

$(sec := sec + 1)$

IncMin action

$$IncMin$$
$$= \qquad \text{[Init schema def]}$$
$$[\Delta State \mid min' = (min + 1)]$$
$$= \qquad \text{[basic conversion } bC \text{ [34]]}$$
$$sec, min : [(inv \land \exists\, sec', min' \bullet min' = (min+1) \land inv'), (inv' \land min' = (min+1))]$$
$$= \qquad \text{[L 5.4 - contract frame [119]]}$$
$$sec : [(inv \land \exists\, sec', min' \bullet min' = (min+1) \land inv')[sec/sec'], (inv' \land min' = (min+1))[sec/sec']]$$
$$= \qquad \text{[predicate calculus]}$$
$$sec : [(\exists\, min' \bullet min' = (min+1)), (min' = (min+1))]$$
$$= \qquad \text{[Assignment conversion } assC \text{ [34], provided } inv[(min+1)/min]]$$
$$(min := min + 1)$$

G.2 Assignment then Choice problem

We want to prove that a state assignment can be shifted to after the choice is resolved. We use the definition of the Ω functions as well as the refinement laws.

G.2.1 Assignment then choice - Both sides

$$\Omega(P_S.((x := 0\,;\; c_1 \to \text{Skip}) \,\square\, (x := 1\,;\; c_2 \to \text{Skip})))$$
$$=$$
$$P_S.((c_1 \to x := 0) \,\square\, (c_2 \to x := 1))$$

Proof

$$\Omega(P_S.((x := 0\,;\; c_1 \to \text{Skip}) \,\square\, (x := 1\,;\; c_2 \to \text{Skip})))$$
$$[\Omega_P]$$
$$= P.$$
$$\textbf{var}\; b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet$$
$$\left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (x := 0\,;\; c_1 \to \text{Skip}) \\ \square \\ (x := 1\,;\; c_2 \to \text{Skip}) \end{array}\right); \\ terminate \to \text{Skip} \end{array}\right) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\ Memory(b) \end{array}\right) \setminus MEM_I$$
$$[\Omega_A\text{-}12]$$
$$= P.$$

$$\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \rightarrow \\ \quad \left(\begin{array}{l} (x := 0\ ;\ \ c_1 \rightarrow \text{Skip}) \\ \Box \\ (x := 1\ ;\ \ c_2 \rightarrow \text{Skip}) \end{array}\right) \end{array}\right); \\ terminate \rightarrow \text{Skip} \end{array}\right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array}\right) \setminus MEM_I$$

[Ω_A 19]

$= P.$

$$\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \rightarrow \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip} \end{array}\right) \\ \Box \\ \left(\begin{array}{l} mset.x.1 \rightarrow \text{Skip}; \\ c_2 \rightarrow \text{Skip} \end{array}\right) \end{array}\right) \end{array}\right); \\ terminate \rightarrow \text{Skip} \end{array}\right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array}\right) \setminus MEM_I$$

[Lemma D.1]

$= P.$

$$\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \rightarrow \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip} \end{array}\right) \\ \Box \\ \left(\begin{array}{l} mset.x.1 \rightarrow \text{Skip}; \\ c_2 \rightarrow \text{Skip} \end{array}\right) \end{array}\right) \end{array}\right); \\ terminate \rightarrow \text{Skip} \end{array}\right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ \left(\begin{array}{l} (\ \Box\, mget.n!b(n) \rightarrow Memory(b)\ ) \\ \Box\,(\ \Box\, mset.n?nv \rightarrow Memory(b \oplus \{n \mapsto nv\})\ ) \\ \Box terminate \rightarrow \text{Skip} \end{array}\right) \end{array}\right) \setminus MEM_I$$

[L.1]

provided:

$\{terminate\} \subseteq MEM_I$

$\{mset, mget\} \subseteq MEM_I$

$mset \in MEM_I \wedge terminate \in MEM_I$

$\{mset, terminate\} \nsubseteq \{mget\}$

$= P.$

$$\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \rightarrow \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip} \end{array}\right) \\ \Box \\ \left(\begin{array}{l} mset.x.1 \rightarrow \text{Skip}; \\ c_2 \rightarrow \text{Skip} \end{array}\right) \end{array}\right) \end{array}\right); \\ terminate \rightarrow \text{Skip} \end{array}\right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ mget.x!vx \rightarrow Memory(b) \end{array}\right) \setminus MEM_I$$

[L.2]

provided:

$\{mget\} \subseteq MEM_I$

$b \notin FV(Memory(b))$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.0 \to \mathrm{Skip}; \\ c_1 \to \mathrm{Skip} \end{array}\right) \\
\Box \\
\left(\begin{array}{l} \mathit{mset}.x.1 \to \mathrm{Skip}; \\ c_2 \to \mathrm{Skip} \end{array}\right)
\end{array}\right); \\
\mathit{terminate} \to \mathrm{Skip}
\end{array}\right) \\
\llbracket \varnothing \mid \mathit{MEM_I} \mid \{b\} \rrbracket\ \mathit{Memory}(b)
\end{array}\right) \setminus \mathit{MEM_I}
\end{array}
$$

[L.4]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(\mathbf{True})\ \&\ \left(\begin{array}{l} \mathit{mset}.x.0 \to \mathrm{Skip}; \\ c_1 \to \mathrm{Skip} \end{array}\right) \\
\Box \\
(\mathbf{True})\ \&\ \left(\begin{array}{l} \mathit{mset}.x.1 \to \mathrm{Skip}; \\ c_2 \to \mathrm{Skip} \end{array}\right)
\end{array}\right); \\
\mathit{terminate} \to \mathrm{Skip}
\end{array}\right) \\
\llbracket \varnothing \mid \mathit{MEM_I} \mid \{b\} \rrbracket\ \mathit{Memory}(b)
\end{array}\right) \setminus \mathit{MEM_I}
\end{array}
$$

[L.5]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left((\mathbf{True})\ \&\ \left(\begin{array}{l} \mathit{mset}.x.0 \to \mathrm{Skip}; \\ c_1 \to \mathrm{Skip}; \\ \mathit{terminate} \to \mathrm{Skip} \end{array}\right)\right) \\
\Box \\
\left((\mathbf{True})\ \&\ \left(\begin{array}{l} \mathit{mset}.x.1 \to \mathrm{Skip}; \\ c_2 \to \mathrm{Skip}; \\ \mathit{terminate} \to \mathrm{Skip} \end{array}\right)\right)
\end{array}\right) \\
\llbracket \varnothing \mid \mathit{MEM_I} \mid \{b\} \rrbracket\ \mathit{Memory}(b)
\end{array}\right) \setminus \mathit{MEM_I}
\end{array}
$$

[L.4]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.0 \to \mathrm{Skip}; \\ c_1 \to \mathrm{Skip}; \\ \mathit{terminate} \to \mathrm{Skip} \end{array}\right) \\
\Box \\
\left(\begin{array}{l} \mathit{mset}.x.1 \to \mathrm{Skip}; \\ c_2 \to \mathrm{Skip}; \\ \mathit{terminate} \to \mathrm{Skip} \end{array}\right)
\end{array}\right) \\
\llbracket \varnothing \mid \mathit{MEM_I} \mid \{b\} \rrbracket\ \mathit{Memory}(b)
\end{array}\right) \setminus \mathit{MEM_I}
\end{array}
$$

[L.6]

provided:

$\mathit{initials}(\mathit{Memory}(b)) \subseteq \mathit{MEM_I}$

$\mathit{Memory}(b)$ is deterministic

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.0 \to \mathrm{Skip}; \\ c_1 \to \mathrm{Skip}; \\ \mathit{terminate} \to \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid \mathit{MEM_I} \mid \{b\} \rrbracket\ \mathit{Memory}(b)
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.1 \to \mathrm{Skip}; \\ c_2 \to \mathrm{Skip}; \\ \mathit{terminate} \to \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid \mathit{MEM_I} \mid \{b\} \rrbracket\ \mathit{Memory}(b)
\end{array}\right)
\end{array}\right) \setminus \mathit{MEM_I}
\end{array}
$$

[Lemma D.1]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\left(\begin{array}{l}
\mathbf{var}\ b : BINDING \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \mathrm{Skip}; \\ c_1 \rightarrow \mathrm{Skip}; \\ terminate \rightarrow \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
\left(\begin{array}{l} (\ \Box\, mget.n!b(n) \rightarrow Memory(b)\ ) \\ \Box\, (\ \Box\, mset.n?nv \rightarrow Memory(b \oplus \{n \mapsto nv\})\ ) \\ \Box terminate \rightarrow \mathrm{Skip} \end{array}\right)
\end{array}\right) \\
\Box \\
\mathbf{var}\ b : BINDING \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.1 \rightarrow \mathrm{Skip}; \\ c_2 \rightarrow \mathrm{Skip}; \\ terminate \rightarrow \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
\left(\begin{array}{l} (\ \Box\, mget.n!b(n) \rightarrow Memory(b)\ ) \\ \Box\, (\ \Box\, mset.n?nv \rightarrow Memory(b \oplus \{n \mapsto nv\})\ ) \\ \Box terminate \rightarrow \mathrm{Skip} \end{array}\right)
\end{array}\right)
\end{array}\right) \setminus MEM_I
\end{array}
$$

[L.1]

provided:

$\{terminate\} \subseteq MEM_I$

$\{mget, mset\} \subseteq MEM_I$

$\{mset, terminate\} \subseteq MEM_I$

$\{mset, terminate\} \not\subseteq \{mset\}$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \mathrm{Skip}; \\ c_1 \rightarrow \mathrm{Skip}; \\ terminate \rightarrow \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
mset.x?0 \rightarrow Memory(b \oplus \{x \mapsto 0\})
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.1 \rightarrow \mathrm{Skip}; \\ c_2 \rightarrow \mathrm{Skip}; \\ terminate \rightarrow \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
mset.x?1 \rightarrow Memory(b \oplus \{x \mapsto 1\})
\end{array}\right)
\end{array}\right) \\
\setminus MEM_I
\end{array}
$$

[L.2+L.3]

provided:

$\{mset\} \subseteq MEM_I$

$x \notin FV(Memory(b))$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} c_1 \rightarrow \mathrm{Skip}; \\ terminate \rightarrow \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
Memory(b \oplus \{x \mapsto 0\})
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} c_2 \rightarrow \mathrm{Skip}; \\ terminate \rightarrow \mathrm{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
Memory(b \oplus \{x \mapsto 1\})
\end{array}\right)
\end{array}\right) \\
\setminus MEM_I
\end{array}
$$

[L.10]

provided:

$initials(Memory(b)) \subseteq MEM_I$
$MEM_I \cap usedC(c_1 \to \text{Skip}) = \varnothing$
$MEM_I \cap usedC(c_2 \to \text{Skip}) = \varnothing$
$wrtV(c_1 \to \text{Skip}) \cap usedV(Memory(b)) = \varnothing$
$wrtV(c_2 \to \text{Skip}) \cap usedV(Memory(b)) = \varnothing$
$Memory(b)$ is divergence free
$wrtV(c_1 \to \text{Skip}) \subseteq \varnothing$
$wrtV(c_2 \to \text{Skip}) \subseteq \varnothing$

$= P.$

$$\textbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (c_1 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \end{array}\right) \\ \square \\ \left(\begin{array}{l} (c_2 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 1\}) \end{array}\right) \end{array}\right) \end{array}\right) \setminus MEM_I$$

[L.7]

$= P.$

$$\textbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (c_1 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \end{array}\right) \setminus MEM_I \\ \square \\ \left(\begin{array}{l} (c_2 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 1\}) \end{array}\right) \end{array}\right) \\ \setminus MEM_I \end{array}\right)$$

[L.8+L.9]

provided:

$MEM_I \cap usedC(c_1 \to \text{Skip}) \cap usedC(c_2 \to \text{Skip}) = \varnothing$

$= P.$

$$\textbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} (c_1 \to \text{Skip}); \\ \left(\left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \setminus MEM_I\right) \\ \square \\ (c_2 \to \text{Skip}); \\ \left(\left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 1\}) \end{array}\right) \setminus MEM_I\right) \end{array}\right)$$

[L.3]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
(c_1 \to \text{Skip}); \\
\left(\left(\begin{array}{l} (\text{Skip}\ ;\ \ terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \setminus MEM_I \right) \\
\Box \\
(c_2 \to \text{Skip}); \\
\left(\left(\begin{array}{l} (\text{Skip}\ ;\ \ terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 1\}) \end{array}\right) \setminus MEM_I \right)
\end{array}\right)
\end{array}
$$

[L.2]

provided:

$\{mset\} \subseteq MEM_I$

$x \notin FV(Memory(b))$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
(c_1 \to \text{Skip}); \\
\left(\begin{array}{l} \left(\begin{array}{l} (mset.x.0 \to \text{Skip}\ ;\ \ terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ mset.x?0 \to Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \\ \setminus MEM_I \end{array}\right) \\
\Box \\
(c_2 \to \text{Skip}); \\
\left(\begin{array}{l} \left(\begin{array}{l} (mset.x.1 \to \text{Skip}\ ;\ \ terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ mgset.x?1 \to Memory(b \oplus \{x \mapsto 1\}) \end{array}\right) \\ \setminus MEM_I \end{array}\right)
\end{array}\right)
\end{array}
$$

[L.1]

provided:

$\{terminate\} \subseteq MEM_I$

$\{mget, mset\} \subseteq MEM_I$

$\{mset, terminate\} \subseteq MEM_I$

$\{mset, terminate\} \nsubseteq \{mset\}$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
(c_1 \to \text{Skip}); \\
\left(\begin{array}{l} \left(\begin{array}{l} (mset.x.0 \to \text{Skip}\ ;\ \ terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ \left(\begin{array}{l} (\ \Box\, mget.n!b(n) \to Memory(b)\ ) \\ \Box\, (\ \Box\, mset.n?nv \to Memory(b \oplus \{n \mapsto nv\})\ ) \\ \Box terminate \to \text{Skip} \end{array}\right) \end{array}\right) \\ \setminus MEM_I \end{array}\right) \\
\Box \\
(c_2 \to \text{Skip}); \\
\left(\begin{array}{l} \left(\begin{array}{l} (mset.x.1 \to \text{Skip}\ ;\ \ terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ \left(\begin{array}{l} (\ \Box\, mget.n!b(n) \to Memory(b)\ ) \\ \Box\, (\ \Box\, mset.n?nv \to Memory(b \oplus \{n \mapsto nv\})\ ) \\ \Box terminate \to \text{Skip} \end{array}\right) \end{array}\right) \\ \setminus MEM_I \end{array}\right)
\end{array}\right)
\end{array}
$$

[Lemma D.1]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
(c_1 \rightarrow \text{Skip}); \\
\left(\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.0 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) \\
[\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!] \\
\mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM_I} \right) \\
\Box \\
(c_2 \rightarrow \text{Skip}); \\
\left(\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.1 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) \\
[\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!] \\
\mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM_I} \right)
\end{array} \right)
\end{array}
$$

[L.9+L.8]

provided:

$\mathit{MEM_I} \cap \mathit{usedC}(c_1 \rightarrow \text{Skip}) \cap \mathit{usedC}(c_2 \rightarrow \text{Skip}) = \varnothing$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
(c_1 \rightarrow \text{Skip})\ ;\ \left(\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.0 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) \\
[\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!] \\
\mathit{Memory}(b)
\end{array} \right) \right) \setminus \mathit{MEM_I} \\
\Box \\
(c_2 \rightarrow \text{Skip})\ ;\ \left(\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.1 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) \\
[\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!] \\
\mathit{Memory}(b)
\end{array} \right) \right) \setminus \mathit{MEM_I}
\end{array} \right)
\end{array}
$$

[L.7]

provided:

$\mathit{initials}(A_1) \cup \mathit{initials}(A_2) \cap \mathit{MEM_I} = \varnothing$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
(c_1 \rightarrow \text{Skip})\ ;\ \left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.0 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) \\
[\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!] \\
\mathit{Memory}(b)
\end{array} \right) \\
\Box \\
(c_2 \rightarrow \text{Skip})\ ;\ \left(\begin{array}{l}
\left(\begin{array}{l} \mathit{mset}.x.1 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) \\
[\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!] \\
\mathit{Memory}(b)
\end{array} \right)
\end{array} \right) \setminus \mathit{MEM_I}
\end{array}
$$

[L.10]

provided:

$\mathit{initials}(\mathit{Memory}(b)) \subseteq \mathit{MEM_I}$

$\mathit{MEM_I} \cap \mathit{usedC}(c_1 \rightarrow \text{Skip}) = \varnothing$

$\mathit{MEM_I} \cap \mathit{usedC}(c_2 \rightarrow \text{Skip}) = \varnothing$

$\mathit{wrtV}(c_1 \rightarrow \text{Skip}) \cap \mathit{usedV}(\mathit{Memory}(b)) = \varnothing$

$\mathit{wrtV}(c_2 \rightarrow \text{Skip}) \cap \mathit{usedV}(\mathit{Memory}(b)) = \varnothing$

$\mathit{Memory}(b)$ is divergence free

$\mathit{wrtV}(c_1 \rightarrow \text{Skip}) \subseteq \varnothing$

$\mathit{wrtV}(c_2 \rightarrow \text{Skip}) \subseteq \varnothing$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \land \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\left(\begin{array}{l} c_1 \rightarrow \text{Skip}; \\ \mathit{mset}.x.0 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) [\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!]\ \mathit{Memory}(b) \right) \\
\Box \\
\left(\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ \mathit{mset}.x.1 \rightarrow \text{Skip}; \\ \mathit{terminate} \rightarrow \text{Skip} \end{array} \right) [\![\varnothing \mid \mathit{MEM_I} \mid \{b\}]\!]\ \mathit{Memory}(b) \right)
\end{array} \right) \\
\setminus \mathit{MEM_I}
\end{array}
$$

[L.6]

provided:
$initials(Memory(b)) \subseteq MEM_I$
$Memory(b)$ is deterministic

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\left(\begin{array}{l} c_1 \to \text{Skip}; \\ mset.x.0 \to \text{Skip}; \\ terminate \to \text{Skip} \end{array} \right) \Box \left(\begin{array}{l} c_2 \to \text{Skip}; \\ mset.x.1 \to \text{Skip}; \\ terminate \to \text{Skip} \end{array} \right) \right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b) \end{array} \right) \setminus MEM_I$$

[L.4]

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\mathbf{True}) \ \& \left(\begin{array}{l} c_1 \to \text{Skip}; \\ mset.x.0 \to \text{Skip}; \\ terminate \to \text{Skip} \end{array} \right) \\ \Box \\ (\mathbf{True}) \ \& \left(\begin{array}{l} c_2 \to \text{Skip}; \\ mset.x.1 \to \text{Skip}; \\ terminate \to \text{Skip} \end{array} \right) \end{array} \right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b) \end{array} \right) \setminus MEM_I$$

[L.5]

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (\mathbf{True}) \ \& \left(\begin{array}{l} c_1 \to \text{Skip}; \\ mset.x.0 \to \text{Skip} \end{array} \right) \\ \Box \\ (\mathbf{True}) \ \& \left(\begin{array}{l} c_2 \to \text{Skip}; \\ mset.x.1 \to \text{Skip} \end{array} \right) \end{array} \right); \\ terminate \to \text{Skip} \end{array} \right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b) \end{array} \right) \setminus MEM_I$$

[L.4]

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (c_1 \to \text{Skip}\ ;\ mset.x.0 \to \text{Skip}) \\ \Box \\ (c_2 \to \text{Skip}\ ;\ mset.x.1 \to \text{Skip}) \end{array} \right); \\ terminate \to \text{Skip} \end{array} \right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b) \end{array} \right) \setminus MEM_I$$

[L.2]

provided:
$\{mget\} \subseteq MEM_I$
$b \notin FV(Memory(b))$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \to \\ \left(\begin{array}{l} (c_1 \to \text{Skip}\ ;\ mset.x.0 \to \text{Skip}) \\ \Box \\ (c_2 \to \text{Skip}\ ;\ mset.x.1 \to \text{Skip}) \end{array} \right) \end{array} \right); \\ terminate \to \text{Skip} \end{array} \right) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ mget.x!vx \to Memory(b) \end{array} \right) \setminus MEM_I$$

[L.1]

provided:
$\{terminate\} \subseteq MEM_I$
$\{mset, mget\} \subseteq MEM_I$
$mset \in MEM_I \wedge terminate \in MEM_I$
$\{mset, terminate\} \nsubseteq \{mget\}$

$$
\begin{array}{l}
= P. \\
\quad \mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \wedge \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{mget}.x?vx \rightarrow \\
\left(\begin{array}{l}
(c_1 \rightarrow \mathrm{Skip}\ ;\ \ \mathit{mset}.x.0 \rightarrow \mathrm{Skip}) \\
\Box \\
(c_2 \rightarrow \mathrm{Skip}\ ;\ \ \mathit{mset}.x.1 \rightarrow \mathrm{Skip})
\end{array} \right)
\end{array} \right) ; \\
\mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right) \\
[\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!] \\
\left(\begin{array}{l}
(\ \Box\, \mathit{mget}.n!b(n) \rightarrow \mathit{Memory}(b)\ ) \\
\Box\, (\ \Box\, \mathit{mset}.n?nv \rightarrow \mathit{Memory}(b \oplus \{n \mapsto nv\})\ ) \\
\Box\, \mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right)
\end{array} \right) \setminus \mathit{MEM}_I
\end{array}
$$

[Lemma D.1]

$$
\begin{array}{l}
= P. \\
\quad \mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \wedge \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{mget}.x?vx \rightarrow \\
\left(\begin{array}{l}
(c_1 \rightarrow \mathrm{Skip}\ ;\ \ \mathit{mset}.x.0 \rightarrow \mathrm{Skip}) \\
\Box \\
(c_2 \rightarrow \mathrm{Skip}\ ;\ \ \mathit{mset}.x.1 \rightarrow \mathrm{Skip})
\end{array} \right)
\end{array} \right) ; \\
\mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right) \\
[\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!]\ \mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM}_I
\end{array}
$$

[L.3]

$$
\begin{array}{l}
= P. \\
\quad \mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \wedge \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{mget}.x?vx \rightarrow \\
\left(\begin{array}{l}
(c_1 \rightarrow \mathit{mset}.x.0 \rightarrow \mathrm{Skip}) \\
\Box \\
(c_2 \rightarrow \mathit{mset}.x.1 \rightarrow \mathrm{Skip})
\end{array} \right)
\end{array} \right) ; \\
\mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right) \\
[\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!]\ \mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM}_I
\end{array}
$$

[Ω_A-19]

$$
\begin{array}{l}
= P. \\
\quad \mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \wedge \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{mget}.x?vx \rightarrow \\
\left(\begin{array}{l}
(c_1 \rightarrow (\Omega_A(x := 0))) \\
\Box \\
(c_2 \rightarrow (\Omega_A(x := 1)))
\end{array} \right) ; \\
\mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right) \\
[\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!]\ \mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM}_I
\end{array}
$$

[Ω'_A-17]

$$
\begin{array}{l}
= P. \\
\quad \mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \wedge \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{mget}.x?vx \rightarrow \\
\left(\begin{array}{l}
(c_1 \rightarrow (\Omega'_A(x := 0))) \\
\Box \\
(c_2 \rightarrow (\Omega'_A(x := 1)))
\end{array} \right) ; \\
\mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right) \\
[\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!]\ \mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM}_I
\end{array}
$$

[Ω'_A-2]

$$
\begin{array}{l}
= P. \\
\quad \mathbf{var}\ b : \{ y : \mathit{BINDING} \mid b(x) \in \mathbb{N} \wedge \mathit{inv}(b(x)) \} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{mget}.x?vx \rightarrow \\
\left(\begin{array}{l}
\Omega'_A(c_1 \rightarrow (x := 0)) \\
\Box \\
\Omega'_A(c_2 \rightarrow (x := 1))
\end{array} \right) ; \\
\mathit{terminate} \rightarrow \mathrm{Skip}
\end{array} \right) \\
[\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!]\ \mathit{Memory}(b)
\end{array} \right) \setminus \mathit{MEM}_I
\end{array}
$$

[Lemma D.1]

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \rightarrow \left(\begin{array}{l} \Omega'_A(c_1 \rightarrow (x := 0)) \\ \Box \\ \Omega'_A(c_2 \rightarrow (x := 1)) \end{array} \right); \\ terminate \rightarrow \text{Skip} \end{array} \right) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\ \left(\begin{array}{l} (\ \Box\, mget.n!b(n) \rightarrow Memory(b)\ ) \\ \Box\, (\ \Box\, mset.n?nv \rightarrow Memory(b \oplus \{n \mapsto nv\})\ ) \\ \Box terminate \rightarrow \text{Skip} \end{array} \right) \end{array} \right) \setminus MEM_I$$

[L.1]

provided:
$\{terminate\} \subseteq MEM_I$
$\{mset, mget\} \subseteq MEM_I$
$mset \in MEM_I \wedge terminate \in MEM_I$
$\{mset, terminate\} \nsubseteq \{mget\}$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} mget.x?vx \rightarrow \left(\begin{array}{l} \Omega'_A(c_1 \rightarrow (x := 0)) \\ \Box \\ \Omega'_A(c_2 \rightarrow (x := 1)) \end{array} \right); \\ terminate \rightarrow \text{Skip} \end{array} \right) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\ mget.n!b(n) \rightarrow Memory(b) \end{array} \right) \setminus MEM_I$$

[L.2]

provided:
$\{mget\} \subseteq MEM_I$
$b \notin FV(Memory(b))$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} \Omega'_A((c_1 \rightarrow (x := 0)) \\ \Box \\ (c_2 \rightarrow (x := 1))) \end{array} \right); \\ terminate \rightarrow \text{Skip} \end{array} \right) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\ mget.n!b(n) \rightarrow Memory(b) \end{array} \right) \setminus MEM_I$$

[L.4+L.6 twice]

provided:
$initials(Memory(b)) \subseteq MEM_I$
$Memory(b)$ is deterministic

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\ \Omega'_A(c_1 \rightarrow (x := 0));\ terminate \rightarrow \text{Skip}\ ) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b) \end{array} \right) \\ \Box \\ \left(\begin{array}{l} (\ \Omega'_A(c_2 \rightarrow (x := 1));\ terminate \rightarrow \text{Skip}\ ) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b) \end{array} \right) \end{array} \right) \setminus MEM_I$$

[L.7]

provided:
$initials(c_1 \rightarrow (x := 0)) \cup initials(c_2 \rightarrow (x := 1)) \cap MEM_I = \varnothing$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\ \Omega'_A(c_1 \rightarrow (x := 0));\ terminate \rightarrow \text{Skip}\ ) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b) \end{array} \right) \setminus MEM_I \\ \Box \\ \left(\begin{array}{l} (\ \Omega'_A(c_2 \rightarrow (x := 1));\ terminate \rightarrow \text{Skip}\ ) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b) \end{array} \right) \setminus MEM_I \end{array} \right)$$

[Induction Hypoteses]

$$= P. \quad \mathbf{var}\, b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \left(\begin{array}{l} (\mathbf{vres}\, k : BINDING \bullet (c_1 \rightarrow (x := 0)))(b) \\ \Box \\ (\mathbf{vres}\, k : BINDING \bullet (c_2 \rightarrow (x := 1)))(b) \end{array}\right)$$

[Semantics]

$$= P. \quad \mathbf{var}\, b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \left(\begin{array}{l} (\mathbf{vres}\, k : BINDING \bullet (c_1 \rightarrow (x := 0)))(b) \\ \Box \\ (\mathbf{vres}\, m : BINDING \bullet (c_2 \rightarrow (x := 1)))(b) \end{array}\right)$$

[L.11]

provided:

$k \notin FV(c_2 \rightarrow (x := 1)) \land m \notin FV(c_1 \rightarrow (x := 0))$

$$= P. \quad \mathbf{var}\, b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \left(\begin{array}{l} (\mathbf{var}\, k : BINDING \bullet k := b \,;\; (c_1 \rightarrow (x := 0)) \,;\; b := k) \\ \Box \\ (\mathbf{var}\, m : BINDING \bullet m := b \,;\; (c_2 \rightarrow (x := 1)) \,;\; b := m) \end{array}\right)$$

[L.12]

provided:

$k \notin FV((c_2 \rightarrow (x := 1))(b)) \land m \notin FV((c_1 \rightarrow (x := 0))(b))$

$$= P. \quad \mathbf{var}\, b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \left(\begin{array}{l} (\mathbf{var}\, k : BINDING \bullet (c_1 \rightarrow (x := 0)) \,;\; b := b) \\ \Box \\ (\mathbf{var}\, m : BINDING \bullet (c_2 \rightarrow (x := 1)) \,;\; b := b) \end{array}\right)$$

[L.13+L.3]

provided:

$k \notin FV((c_2 \rightarrow (x := 1))(b)) \land m \notin FV((c_1 \rightarrow (x := 0))(b))$

$$= P. \quad \mathbf{var}\, b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \left(\begin{array}{l} (\mathbf{var}\, k : BINDING \bullet c_1 \rightarrow (x := 0)) \\ \Box \\ (\mathbf{var}\, m : BINDING \bullet c_2 \rightarrow (x := 1)) \end{array}\right)$$

[L.14]

provided:

$k \notin FV(c_2 \rightarrow (x := 1)) \land m \notin FV(c_1 \rightarrow (x := 0))$

$$= P. \quad \mathbf{var}\, b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet ((c_1 \rightarrow (x := 0)) \Box (c_2 \rightarrow (x := 1)))$$

[L.20]

$$= P.((c_1 \rightarrow (x := 0)) \Box (c_2 \rightarrow (x := 1)))$$

G.2.2 Assignment then choice - Assignment in one side

$$\Omega(P_S.((x := 0 \,;\; c_1 \rightarrow \text{Skip}) \Box (c_2 \rightarrow \text{Skip})))$$
$$=$$
$$\Omega(P_S.((c_1 \rightarrow x := 0) \Box (c_2 \rightarrow \text{Skip})))$$

Proof

$$\Omega(P_S.((x := 0 \,;\; c_1 \rightarrow \text{Skip}) \Box (c_2 \rightarrow \text{Skip})))$$

$$[\Omega_P]$$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}(x := 0\ ;\ \ c_1 \to \text{Skip})\\ \Box\\ (c_2 \to \text{Skip})\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right) \setminus MEM_I$$

$$[\Omega_A - \Box]$$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}(x := 0\ ;\ \ c_1 \to \text{Skip})\\ \Box\\ (c_2 \to \text{Skip})\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right) \setminus MEM_I$$

$$[\Omega_A - :=]$$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}\left(\begin{array}{l}mset.x.0 \to \text{Skip};\\ c_1 \to \text{Skip}\end{array}\right)\\ \Box\\ (\ \ c_2 \to \text{Skip}\ \ )\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right) \setminus MEM_I$$

[Lemma K.2 from Deliverable 24.1]

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}\left(\begin{array}{l}mset.x.0 \to \text{Skip};\\ c_1 \to \text{Skip}\end{array}\right)\\ \Box\\ (\ c_2 \to \text{Skip}\ )\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ \left(\begin{array}{l}(\ \Box\, mget.n!b(n) \to Memory(b)\ )\\ \Box\,(\ \Box\, mset.n?nv \to Memory(b \oplus \{n \mapsto nv\})\ )\\ \Box terminate \to \text{Skip}\end{array}\right)\end{array}\right) \setminus MEM_I$$

[L.1]

provided:
$\{terminate\} \subseteq MEM_I$
$\{mset, mget\} \subseteq MEM_I$
$mset \in MEM_I \wedge terminate \in MEM_I$
$\{mset, terminate\} \subsetneq \{mget\}$

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
mget.x?vx \rightarrow \\
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip} \end{array}\right) \\
\Box \\
\left(c_2 \rightarrow \text{Skip} \right)
\end{array}\right)
\end{array}\right) ; \\
terminate \rightarrow \text{Skip}
\end{array}\right) \\
[\![\varnothing \mid MEM_I \mid \{b\}]\!] \\
mget.x!vx \rightarrow Memory(b)
\end{array}\right) \setminus MEM_I
\end{array}
$$

[L.2]

provided:
$\{mget\} \subseteq MEM_I$ $\qquad = P.$
$b \notin FV(Memory(b))$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip} \end{array}\right) \\
\Box \\
\left(c_2 \rightarrow \text{Skip} \right)
\end{array}\right) ; \\
terminate \rightarrow \text{Skip}
\end{array}\right) \\
[\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b)
\end{array}\right) \setminus MEM_I
\end{array}
$$

[L.4]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(\mathbf{True}) \& \\
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip} \end{array}\right) \\
\Box \\
(\mathbf{True}) \& \\
\left(c_2 \rightarrow \text{Skip} \right)
\end{array}\right) ; \\
terminate \rightarrow \text{Skip}
\end{array}\right) \\
[\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b)
\end{array}\right) \setminus MEM_I
\end{array}
$$

[L.5]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
(\mathbf{True}) \& \\
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right)
\end{array}\right) \Box \\
\left(\begin{array}{l}
(\mathbf{True}) \& \\
\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right)
\end{array}\right)
\end{array}\right) \\
[\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b)
\end{array}\right) \setminus MEM_I
\end{array}
$$

[L.4]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\Box \\
\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right)
\end{array}\right) \\
[\![\varnothing \mid MEM_I \mid \{b\}]\!]\ Memory(b)
\end{array}\right) \setminus MEM_I
\end{array}
$$

[L.6]

provided:
$initials(Memory(b)) \subseteq MEM_I$
$Memory(b)$ is deterministic
$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b)
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b)
\end{array}\right)
\end{array}\right) \\
\quad \setminus MEM_I
\end{array}
$$

[Lemma K.2 Deliverable 24.1]

$= P.$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\mathbf{var}\ b : BINDING \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
\left(\begin{array}{l}
\left(\ \Box\, mget.n!b(n) \rightarrow Memory(b) \ \right) \\
\Box \left(\ \Box\, mset.n?nv \rightarrow Memory(b \oplus \{n \mapsto nv\}) \ \right) \\
\Box terminate \rightarrow \text{Skip}
\end{array}\right)
\end{array}\right) \\
\Box \\
\mathbf{var}\ b : BINDING \bullet \\
\left(\begin{array}{l}
\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b)
\end{array}\right)
\end{array}\right) \\
\quad \setminus MEM_I
\end{array}
$$

[L.1]

provided:
$\{terminate\} \subseteq MEM_I$
$\{mget, mset\} \subseteq MEM_I$ $\quad = P.$
$\{mset, terminate\} \subseteq MEM_I$
$\{mset, terminate\} \subsetneq \{mset\}$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} mset.x.0 \rightarrow \text{Skip}; \\ c_1 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
mset.x?0 \rightarrow Memory(b \oplus \{x \mapsto 0\})
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b)
\end{array}\right)
\end{array}\right) \\
\quad \setminus MEM_I
\end{array}
$$

[L.2+L.3]

provided:
$\{mset\} \subseteq MEM_I$ $\quad = P.$
$x \notin FV(Memory(b))$

$$
\begin{array}{l}
\mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \land inv(b(x))\} \bullet \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} c_1 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
Memory(b \oplus \{x \mapsto 0\})
\end{array}\right) \\
\Box \\
\left(\begin{array}{l}
\left(\begin{array}{l} c_2 \rightarrow \text{Skip}; \\ terminate \rightarrow \text{Skip} \end{array}\right) \\
\llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \\
Memory(b)
\end{array}\right)
\end{array}\right) \\
\quad \setminus MEM_I
\end{array}
$$

[L.10]

provided:
$initials(Memory(b)) \subseteq MEM_I$
$MEM_I \cap usedC(c_1 \to \text{Skip}) = \varnothing$
$MEM_I \cap usedC(c_2 \to \text{Skip}) = \varnothing$
$wrtV(c_1 \to \text{Skip}) \cap usedV(Memory(b)) = \varnothing$
$wrtV(c_2 \to \text{Skip}) \cap usedV(Memory(b)) = \varnothing$
$Memory(b)$ is divergence free
$wrtV(c_1 \to \text{Skip}) \subseteq \varnothing$
$wrtV(c_2 \to \text{Skip}) \subseteq \varnothing$

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (c_1 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \end{array}\right) \\ \Box \\ \left(\begin{array}{l} (c_2 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array}\right) \end{array}\right) \end{array}\right) \setminus MEM_I$$

[L.7]

$$= P.\ \mathbf{var}\ b : \{y : BINDING \mid b(x) \in \mathbb{N} \wedge inv(b(x))\} \bullet \left(\begin{array}{l} \left(\begin{array}{l} (c_1 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \end{array}\right) \setminus MEM_I \\ \Box \\ \left(\begin{array}{l} (c_2 \to \text{Skip}); \\ \left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array}\right) \end{array}\right) \\ \setminus MEM_I \end{array}\right)$$

[L.8+L.9]

provided:
$MEM_I \cap usedC(c_1 \to \text{Skip}) \cap usedC(c_2 \to \text{Skip}) = \varnothing$

$$= P.b : BINDING \bullet \left(\begin{array}{l} (c_1 \to \text{Skip}); \\ \left(\left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b \oplus \{x \mapsto 0\}) \end{array}\right) \setminus MEM_I\right) \\ \Box \\ (c_2 \to \text{Skip}); \\ \left(\left(\begin{array}{l} (terminate \to \text{Skip}) \\ [\![\varnothing \mid MEM_I \mid \{b\}]\!] \\ Memory(b) \end{array}\right) \setminus MEM_I\right) \end{array}\right)$$

[L.3]

$= P.b : BINDING \bullet$

$$\left(\begin{array}{l}(c_1 \to \text{Skip});\\ \left(\left(\begin{array}{l}(\text{Skip}\ ;\ \ terminate \to \text{Skip})\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b \oplus \{x \mapsto 0\})\end{array}\right) \setminus MEM_I\right)\\ \Box\\ (c_2 \to \text{Skip});\\ \left(\left(\begin{array}{l}(terminate \to \text{Skip})\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right) \setminus MEM_I\right)\end{array}\right)$$

[L.2]

provided:
$\{mset\} \subseteq MEM_I$
$x \notin FV(Memory(b))$

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}(c_1 \to \text{Skip});\\ \left(\begin{array}{l}\left(\begin{array}{l}(mset.x.0 \to \text{Skip}\ ;\ \ terminate \to \text{Skip})\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ mset.x?0 \to Memory(b \oplus \{x \mapsto 0\})\end{array}\right)\\ \setminus MEM_I\end{array}\right)\\ \Box\\ (c_2 \to \text{Skip});\\ \left(\begin{array}{l}\left(\begin{array}{l}(terminate \to \text{Skip})\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right)\\ \setminus MEM_I\end{array}\right)\end{array}\right)$$

[L.1]

provided:
$\{terminate\} \subseteq MEM_I$
$\{mget, mset\} \subseteq MEM_I$ $\quad = P.b : BINDING \bullet$
$\{mset, terminate\} \subseteq MEM_I$
$\{mset, terminate\} \subsetneq \{mset\}$

$$\left(\begin{array}{l}(c_1 \to \text{Skip});\\ \left(\begin{array}{l}\left(\begin{array}{l}(mset.x.0 \to \text{Skip}\ ;\ \ terminate \to \text{Skip})\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ \left(\begin{array}{l}(\ \Box\, mget.n!b(n) \to Memory(b)\ )\\ \Box\,(\ \Box\, mset.n?nv \to Memory(b \oplus \{n \mapsto nv\})\ )\\ \Box terminate \to \text{Skip}\end{array}\right)\end{array}\right)\\ \setminus MEM_I\end{array}\right)\\ \Box\\ (c_2 \to \text{Skip});\\ \left(\begin{array}{l}\left(\begin{array}{l}(terminate \to \text{Skip})\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right)\\ \setminus MEM_I\end{array}\right)\end{array}\right)$$

[Lemma K.2 from Deliverable 24.1]

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}(c_1 \to \text{Skip});\\ \left(\left(\begin{array}{l}\left(\begin{array}{l}mset.x.0 \to \text{Skip};\\ terminate \to \text{Skip}\end{array}\right)\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right) \setminus MEM_I\right)\\ \Box\\ (c_2 \to \text{Skip});\\ \left(\left(\begin{array}{l}(\ terminate \to \text{Skip}\ )\\ [\![\varnothing \mid MEM_I \mid \{b\}]\!]\\ Memory(b)\end{array}\right) \setminus MEM_I\right)\end{array}\right)$$

[L.9+L.8]

provided:
$MEM_I \cap usedC(c_1 \to \text{Skip}) \cap usedC(c_2 \to \text{Skip}) = \varnothing$

$= P.b : \mathit{BINDING} \bullet$

$$\left(\begin{array}{l} (c_1 \to \text{Skip}) \; ; \left(\left(\begin{array}{l}\left(\begin{array}{l} \mathit{mset}.x.0 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right) \\ [\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!] \\ \mathit{Memory}(b)\end{array}\right)\right) \setminus \mathit{MEM}_I \\ \Box \\ (c_2 \to \text{Skip}) \; ; \left(\left(\begin{array}{l} (\ \mathit{terminate} \to \text{Skip}\ ) \\ [\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!] \\ \mathit{Memory}(b)\end{array}\right)\right) \setminus \mathit{MEM}_I \end{array}\right)$$

[L.7]

provided:
$\mathit{initials}(c_1 \to \text{Skip}) \cup \mathit{initials}(c_2 \to \text{Skip}) \cap \mathit{MEM}_I = \varnothing$

$= P.b : \mathit{BINDING} \bullet$

$$\left(\begin{array}{l} (c_1 \to \text{Skip}) \; ; \left(\begin{array}{l}\left(\begin{array}{l} \mathit{mset}.x.0 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right) \\ [\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!] \\ \mathit{Memory}(b)\end{array}\right) \\ \Box \\ (c_2 \to \text{Skip}) \; ; \left(\begin{array}{l} (\ \mathit{terminate} \to \text{Skip}\ ) \\ [\![\varnothing \mid \mathit{MEM}_I \mid \{b\}]\!] \\ \mathit{Memory}(b)\end{array}\right) \end{array}\right) \setminus \mathit{MEM}_I$$

[L.10]

provided:
$\mathit{initials}(\mathit{Memory}(b)) \subseteq \mathit{MEM}_I$
$\mathit{MEM}_I \cap \mathit{usedC}(c_1 \to \text{Skip}) = \varnothing$
$\mathit{wrtV}(c_1 \to \text{Skip}) \cap \mathit{usedV}(\mathit{Memory}(b)) = \varnothing$
$\mathit{Memory}(b)$ is divergence free
$\mathit{wrtV}(c_1 \to \text{Skip}) \subseteq \varnothing$

$= P.b : \mathit{BINDING} \bullet$

$$\left(\begin{array}{l} \left(\left(\begin{array}{l} c_1 \to \text{Skip}; \\ \mathit{mset}.x.0 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right) [\![\ \varnothing \mid \mathit{MEM}_I \mid \{b\}\ ]\!]\ \mathit{Memory}(b)\right) \\ \Box \\ \left(\left(\begin{array}{l} c_2 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right) [\![\ \varnothing \mid \mathit{MEM}_I \mid \{b\}\ ]\!]\ \mathit{Memory}(b)\right) \end{array}\right) \setminus \mathit{MEM}_I$$

[L.6]

provided:
$\mathit{initials}(\mathit{Memory}(b)) \subseteq \mathit{MEM}_I$
$\mathit{Memory}(b)$ is deterministic

$= P.b : \mathit{BINDING} \bullet$

$$\left(\begin{array}{l} \left(\left(\begin{array}{l} c_1 \to \text{Skip}; \\ \mathit{mset}.x.0 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right) \Box \left(\begin{array}{l} c_2 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right)\right) \\ [\![\varnothing \mid \mathit{MEM}_I \mid \{b\}\ ]\!]\ \mathit{Memory}(b) \end{array}\right) \setminus \mathit{MEM}_I$$

[L.4]

$= P.b : \mathit{BINDING} \bullet$

$$\left(\begin{array}{l} \left(\begin{array}{l} (\mathbf{True}) \;\&\; \left(\begin{array}{l} c_1 \to \text{Skip}; \\ \mathit{mset}.x.0 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right) \\ \Box(\mathbf{True}) \;\&\; \left(\begin{array}{l} c_2 \to \text{Skip}; \\ \mathit{terminate} \to \text{Skip}\end{array}\right)\end{array}\right) \\ [\![\varnothing \mid \mathit{MEM}_I \mid \{b\}\ ]\!]\ \mathit{Memory}(b) \end{array}\right) \setminus \mathit{MEM}_I$$

[L.5]

$= P.b : \mathit{BINDING} \bullet$

$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}(\mathbf{True}) \ \& \left(\begin{array}{l}c_1 \to \text{Skip};\\ mset.x.0 \to \text{Skip}\end{array}\right)\\ \Box(\mathbf{True}) \ \& \left(\ c_2 \to \text{Skip}\ \right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \ Memory(b)\end{array}\right) \setminus MEM_I$$

[L.4]

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}(c_1 \to \text{Skip}\ ;\ \ mset.x.0 \to \text{Skip})\\ \Box(c_2 \to \text{Skip})\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \ Memory(b)\end{array}\right) \setminus MEM_I$$

[L.2]

provided:
$\{mget\} \subseteq MEM_I$
$b \notin FV(Memory(b))$

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}(c_1 \to \text{Skip}\ ;\ \ mset.x.0 \to \text{Skip})\\ \Box(c_2 \to \text{Skip})\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\\ mget.x!vx \to Memory(b)\end{array}\right) \setminus MEM_I$$

[L.1]

provided:
$\{terminate\} \subseteq MEM_I$
$\{mset, mget\} \subseteq MEM_I$
$mset \in MEM_I \land terminate \in MEM_I$
$\{mset, terminate\} \subsetneq \{mget\}$

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}(c_1 \to \text{Skip}\ ;\ \ mset.x.0 \to \text{Skip})\\ \Box(c_2 \to \text{Skip})\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\\ \left(\begin{array}{l}\left(\ \Box\, mget.n!b(n) \to Memory(b)\ \right)\\ \Box \left(\ \Box\, mset.n?nv \to Memory(b \oplus \{n \mapsto nv\})\ \right)\\ \Box terminate \to \text{Skip}\end{array}\right)\end{array}\right) \setminus MEM_I$$

[Lemma K.2 from Deliverable 24.1]

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}(c_1 \to \text{Skip}\ ;\ \ mset.x.0 \to \text{Skip})\\ \Box(c_2 \to \text{Skip})\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \ Memory(b)\end{array}\right) \setminus MEM_I$$

[L.3]

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\begin{array}{l}(c_1 \to mset.x.0 \to \text{Skip})\\ \Box(c_2 \to \text{Skip})\end{array}\right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \ Memory(b)\end{array}\right) \setminus MEM_I$$

$[\Omega_A - :=]$

$$= P.b : BINDING \bullet$$
$$\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}mget.x?vx \to\\ \left(\ (c_1 \to (x := 0)) \Box (c_2 \to \text{Skip})\ \right)\end{array}\right);\\ terminate \to \text{Skip}\end{array}\right)\\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket \ Memory(b)\end{array}\right) \setminus MEM_I$$

$$[\Omega_A - \Box]$$

$$= P.b : BINDING \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\ (c_1 \rightarrow (x := 0)) \Box (c_2 \rightarrow \text{Skip})\ ) ; \\ terminate \rightarrow \text{Skip} \end{array} \right) \\ \llbracket \varnothing \mid MEM_I \mid \{b\} \rrbracket\ Memory(b) \end{array} \right) \setminus MEM_I$$

$$[\Omega_P]$$

$$= \Omega(P_S.((c_1 \rightarrow x := 0) \Box (c_2 \rightarrow \text{Skip})))$$

Appendix H

Chronometer models

We list a number of models derived from a simple example from Oliveira's PhD thesis [129]: a chronometer. First, we present the original models, then we present the results from the translator of Z schemas. Next, we present the state-poor *Circus* specification. Finally, we present the CSP_M code.

H.1 Original models

AChrono

$RANGE == 0 \mathinner{\ldotp\ldotp} 3$
channel $tick, time$
channel $out : \{min, sec : RANGE \bullet (min, sec)\}$
process $AChrono \mathrel{\widehat=}$
begin
  state $AState \mathrel{\widehat=} [sec, min : RANGE]$
  $AInit \mathrel{\widehat=} [AState' \mid sec' = 0;\ min' = 0]$
  $IncSec \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60]$
  $IncMin \mathrel{\widehat=} [\Delta AState \mid min' = (min + 1) \bmod 60]$
  $Run \mathrel{\widehat=}$

$$\left(\begin{array}{l}\left(\begin{array}{l} tick \rightarrow IncSec; \\ \left(\begin{array}{l}\textbf{if}\ sec = 0 \longrightarrow IncMin \\ \,[\!]\ sec \neq 0 \longrightarrow \mathrm{Skip}\ \textbf{fi}\end{array}\right)\end{array}\right) \\ \Box\ time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip}\end{array}\right)$$

  $\bullet\ (AInit\ ;\ (\mu X \bullet (Run\ ;\ X)))$
end

Chrono

channel $inc, minsReq$
channel $ans : RANGE$
channelset $Sync == \{\!|\ inc, minsReq, ans\ |\!\}$

$$
\begin{array}{l}
\textbf{process}\ Chrono \mathrel{\widehat=} \ \textbf{begin} \\
\quad \textbf{state}\ State \mathrel{\widehat=} [sec : RANGE] \land [min : RANGE] \\
\quad SecInit \mathrel{\widehat=} [AState' \mid sec' = 0] \\
\quad IncSec \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
\quad RunSec \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} tick \rightarrow IncSec; \\ \left(\begin{array}{l} \textbf{if}\ sec = 0 \longrightarrow inc \rightarrow \text{Skip} \\ [\!] sec \neq 0 \longrightarrow \text{Skip}\ \textbf{fi} \end{array} \right) \end{array} \right) \\ \Box time \rightarrow minsReq \rightarrow ans?min \rightarrow out!(min, sec) \rightarrow \text{Skip} \end{array} \right) \\
\quad Seconds \mathrel{\widehat=} SecInit\ ;\ (\mu X \bullet (RunSec\ ;\ X)) \\
\quad MinInit \mathrel{\widehat=} [AState' \mid sec' = 0] \\
\quad IncMin \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
\quad RunMin \mathrel{\widehat=} (inc \rightarrow IncMin) \Box (minsReq \rightarrow ans!min \rightarrow \text{Skip}) \\
\quad Minutes \mathrel{\widehat=} MinInit\ ;\ (\mu X \bullet (RunMin\ ;\ X)) \\
\bullet ((Seconds \llbracket \{sec\} \mid Sync \mid \{min\} \rrbracket Minutes) \setminus Sync) \\
\textbf{end}
\end{array}
$$

Distributed processes - *ChronometerFull*

Seconds

$$
\begin{array}{l}
\textbf{process}\ Seconds \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ StateSeconds \mathrel{\widehat=} [sec : RANGE] \\
\quad SecInit \mathrel{\widehat=} [AState' \mid sec' = 0] \\
\quad IncSec \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
\quad RunSec \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} tick \rightarrow IncSec; \\ \left(\begin{array}{l} \textbf{if}\ sec = 0 \longrightarrow inc \rightarrow \text{Skip} \\ [\!] sec \neq 0 \longrightarrow \text{Skip}\ \textbf{fi} \end{array} \right) \end{array} \right) \\ \Box\ time \rightarrow minsReq \rightarrow ans?min \rightarrow out!(min, sec) \rightarrow \text{Skip} \end{array} \right) \\
\quad \bullet\ SecInit\ ;\ (\mu X \bullet (RunSec\ ;\ X)) \\
\textbf{end}
\end{array}
$$

Minutes

$$
\begin{array}{l}
\textbf{process}\ Minutes \mathrel{\widehat=} \\
\ \textbf{begin} \\
\quad \textbf{state}\ StateMinutes \mathrel{\widehat=} [min : RANGE] \\
\quad MinInit \mathrel{\widehat=} [AState' \mid sec' = 0] \\
\quad IncMin \mathrel{\widehat=} [\Delta AState \mid sec' = (sec + 1) \bmod 60] \\
\quad RunMin \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} (inc \rightarrow IncMin) \\ \Box (minsReq \rightarrow ans!min \rightarrow \text{Skip}) \end{array} \right) \\
\quad \bullet\ MinInit\ ;\ (\mu X \bullet (RunMin\ ;\ X)) \\
\textbf{end}
\end{array}
$$

ChronometerFull

$$
\textbf{process}\ ChronometerFull \mathrel{\widehat=} ((Seconds \llbracket Sync \rrbracket Minutes) \setminus Sync)
$$

H.2 After the Z Schemas Translation

AChrono

A specification:

$RANGE == 0 \mathrel{..} 3$
channel $tick, time$
channel $out : \{min, sec : RANGE \bullet (min, sec)\}$
process $AChrono \mathrel{\widehat=}$
begin
state $AState \mathrel{\widehat=} [sec, min : RANGE]$
$AInit \mathrel{\widehat=} (sec, min := 0, 0)$
$IncSec \mathrel{\widehat=} (sec, min := (sec + 1) \bmod 3, min)$
$IncMin \mathrel{\widehat=} (min, sec := (min + 1) \bmod 3, sec)$
$Run \mathrel{\widehat=}$

$$\left(\begin{array}{l} \left(\begin{array}{l} tick \to IncSec; \\ \left(\begin{array}{l} \mathbf{if}\ sec = 0 \longrightarrow IncMin \\ [\!]\, sec \neq 0 \longrightarrow \mathrm{Skip}\ \mathbf{fi} \end{array} \right) \end{array} \right) \\ \Box\ time \to out!(min, sec) \to \mathrm{Skip} \end{array} \right)$$

$\bullet\ (AInit\ ;\ \ (\mu X \bullet (Run\ ;\ \ X)))$
end

Chrono

channel $inc, minsReq$
channel $ans : RANGE$
channelset $Sync == \{\!|\ inc, minsReq, ans\ |\!\}$
process $Chrono \mathrel{\widehat=}$ **begin**
state $State \mathrel{\widehat=} [sec : RANGE;\ min : RANGE]$
$SecInit \mathrel{\widehat=} (sec := 0)$
$IncSec \mathrel{\widehat=} (sec := (sec + 1) \bmod 3)$
$RunSec \mathrel{\widehat=}$

$$\left(\begin{array}{l} \left(\begin{array}{l} tick \to IncSec; \\ \left(\begin{array}{l} \mathbf{if}\ sec = 0 \longrightarrow inc \to \mathrm{Skip} \\ [\!]\, sec \neq 0 \longrightarrow \mathrm{Skip}\ \mathbf{fi} \end{array} \right) \end{array} \right) \\ \Box time \to minsReq \to ans?min \to out!(min, sec) \to \mathrm{Skip} \end{array} \right)$$

$Seconds \mathrel{\widehat=} SecInit\ ;\ \ (\mu X \bullet (RunSec\ ;\ \ X))$
$MinInit \mathrel{\widehat=} (min := 0)$
$IncMin \mathrel{\widehat=} (min := (min + 1) \bmod 3)$
$RunMin \mathrel{\widehat=} (inc \to IncMin) \Box (minsReq \to ans!min \to \mathrm{Skip})$
$Minutes \mathrel{\widehat=} MinInit\ ;\ \ (\mu X \bullet (RunMin\ ;\ \ X))$
$\bullet\ ((Seconds \mathbin{[\![} \{sec\} \mid Sync \mid \{min\} \mathbin{]\!]} Minutes) \setminus Sync)$
end

Distributed processes - *ChronometerFull*

Seconds

$$
\begin{array}{l}
\textbf{process}\ Seconds \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ StateSeconds \mathrel{\widehat=} [sec : RANGE] \\
\quad SecInit \mathrel{\widehat=} (sec := 0) \\
\quad IncSec \mathrel{\widehat=} (sec := (sec + 1) \bmod 3) \\
\quad RunSec \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} \left(\begin{array}{l} tick \rightarrow IncSec; \\ \left(\begin{array}{l} \textbf{if}\ sec = 0 \longrightarrow inc \rightarrow \mathrm{Skip} \\ \talloblong\, sec \neq 0 \longrightarrow \mathrm{Skip}\ \textbf{fi} \end{array} \right) \end{array} \right) \\ \Box\, time \rightarrow minsReq \rightarrow ans?min \rightarrow out!(min, sec) \rightarrow \mathrm{Skip} \end{array} \right) \\
\quad \bullet\ SecInit\ ;\ (\mu X \bullet (RunSec\ ;\ X)) \\
\textbf{end}
\end{array}
$$

Minutes

$$
\begin{array}{l}
\textbf{process}\ Minutes \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ StateMinutes \mathrel{\widehat=} [min : RANGE] \\
\quad MinInit \mathrel{\widehat=} (min := 0) \\
\quad IncMin \mathrel{\widehat=} (min := (min + 1) \bmod 3) \\
\quad RunMin \mathrel{\widehat=} \\
\qquad \left(\begin{array}{l} (inc \rightarrow IncMin) \\ \Box (minsReq \rightarrow ans!min \rightarrow \mathrm{Skip}) \end{array} \right) \\
\quad \bullet\ MinInit\ ;\ (\mu X \bullet (RunMin\ ;\ X)) \\
\textbf{end}
\end{array}
$$

ChronometerFull

$$\textbf{process}\ ChronometerFull \mathrel{\widehat=} ((Seconds\ [\![\ Sync\ ]\!]\ Minutes) \setminus Sync)$$

H.3 CSP_M code from *Circus2CSP*

```
maxValue = 3
maxRange = maxValue - 1
RANGE = {0..maxRange}
channel tick, time
channel out : (RANGE,RANGE)
channel inc, minsReq
channel ans : RANGE
datatype DIRECTION = LEFT | RIGHT

Sync = {| inc,minsReq,ans |}

--------------------------------
-- The universe of values
 datatype UNIVERSE = RAN.RANGE

--Conversions
valueRAN(RAN.v) = v
typeRAN(x) = U_RAN
tagRAN(x) = RAN

-- subtypes of UNIVERSE for RAN
subtype U_RAN = RAN.RANGE

-- definition of NAME for the entire spec
datatype NAME = sv_sec | sv_min

-- Subtype definition for RAN
```

```
b_RAN1 = {(sv_sec, RAN.0),(sv_min, RAN.0)}
subtype NAME_RAN = sv_sec | sv_min
NAMES_VALUES_RAN = seq({seq({(n,v) | v <- typeRAN(n)}) | n <- NAME_RAN})

-- Bindings definitions for RAN
BINDINGS_RAN = {set(b) | b <- set(distCartProd(NAMES_VALUES))}
NAMES_VALUES = seq({seq({(n,v) | v <- typeRAN(n)}) | n <- NAME})

-- Bindings definitions for RAN
BINDINGS = {set(b) | b <- set(distCartProd(NAMES_VALUES))}

--------------------------------
 -- mget, mset and terminate --
--------------------------------
channel mget, mset : NAME.UNIVERSE
channel terminate
MEMI = {| mset,mget,terminate |}

--------------------------------
 -- lget, lset and lterminate --
--------------------------------
channel lget, lset : NAME.UNIVERSE
channel lterminate
MEML = {| lset,lget,lterminate |}
```

```
Minutes(b_RAN) =
  let
     MemoryRANVar(n,b_RAN) =
        ( ( mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
      [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv)))
      [] terminate -> SKIP)
     MemoryRAN(b_RAN) =
        ( [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN) )
     Memory(b_RAN) =
        MemoryRAN(b_RAN)
     MemoryMergeRANVar(n,b_RAN,ns) =
        ( ( lget.n.apply(b_RAN,n) -> MemoryMergeRANVar(n,b_RAN,ns)
      [] lset.n?nv:typeRAN(n) ->F MemoryMergeRANVar(n,over(b_RAN,n,nv),ns))
      [] lterminate ->
    (  ; bd : <b_RAN> @   ; n : <y | y <- ns,member(y,dom(bd))> @  mset.n.apply(bd,n) -> SKIP ))
     MemoryMergeRAN(b_RAN,ns) =
        ( [| {| lterminate |} |] n : dom(b_RAN) @ MemoryMergeRANVar(n,b_RAN,ns) )
     MemoryMerge(b_RAN,ns) =
        MemoryMergeRAN(b_RAN,ns)

  within ( ( ( mset.sv_min.(RAN.0) ->
    ( let X = mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    ( ( inc ->
    mset.sv_min.(RAN.(valueRAN(v_sv_min) + 1) % 3) -> SKIP
      [] minsReq ->
    ans.valueRAN(v_sv_min) -> SKIP);
      X ) within X );
      terminate -> SKIP )
      [| MEMI |] Memory(b_RAN))\MEMI )
```

```
Seconds(b_RAN) =
  let
     MemoryRANVar(n,b_RAN) =
        ( ( mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
      [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv)))
      [] terminate -> SKIP)
     MemoryRAN(b_RAN) =
        ( [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN) )
     Memory(b_RAN) =
        MemoryRAN(b_RAN)
     MemoryMergeRANVar(n,b_RAN,ns) =
        ( ( lget.n.apply(b_RAN,n) -> MemoryMergeRANVar(n,b_RAN,ns)
      [] lset.n?nv:typeRAN(n) -> MemoryMergeRANVar(n,over(b_RAN,n,nv),ns))
      [] lterminate ->
    (  ; bd : <b_RAN> @   ; n : <y | y <- ns,member(y,dom(bd))> @  mset.n.apply(bd,n) -> SKIP ))
     MemoryMergeRAN(b_RAN,ns) =
        ( [| {| lterminate |} |] n : dom(b_RAN) @ MemoryMergeRANVar(n,b_RAN,ns) )
     MemoryMerge(b_RAN,ns) =
        MemoryMergeRAN(b_RAN,ns)

  within ( ( ( mset.sv_sec.(RAN.0) ->
    ( let X = mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( tick ->
    mset.sv_sec.(RAN.(valueRAN(v_sv_sec) + 1) % 3) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ((valueRAN(v_sv_sec) == 0) &
         inc -> SKIP [] (valueRAN(v_sv_sec) != 0) &
```

```
        SKIP)
      [] time ->
    minsReq ->
    ans?t_sv_min ->
    mset.sv_min.(RAN.t_sv_min) ->
    out.(valueRAN(v_sv_min),valueRAN(v_sv_sec)) -> SKIP);
      X ) within X );
      terminate -> SKIP )
      [| MEMI |] Memory(b_RAN))\MEMI )
```

```
ChronometerFullAOG(b) = ( ( SecondsAOG(b) [| Sync |] MinutesAOG(b) ) \ Sync )
```

```
Chrono(b_RAN) =
  let
     MemoryRANVar(n,b_RAN) =
        ( ( mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
      [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv)))
      [] terminate -> SKIP)
     MemoryRAN(b_RAN) =
        ( [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN) )
     Memory(b_RAN) =
        MemoryRAN(b_RAN)
     MemoryMergeRANVar(n,b_RAN,ns) =
        ( ( lget.n.apply(b_RAN,n) -> MemoryMergeRANVar(n,b_RAN,ns)
      [] lset.n?nv:typeRAN(n) -> MemoryMergeRANVar(n,over(b_RAN,n,nv),ns))
      [] lterminate ->
    (  ; bd : <b_RAN> @   ; n : <y | y <- ns,member(y,dom(bd))> @  mset.n.apply(bd,n) -> SKIP ))
     MemoryMergeRAN(b_RAN,ns) =
        ( [| {| lterminate |} |] n : dom(b_RAN) @ MemoryMergeRANVar(n,b_RAN,ns) )
     MemoryMerge(b_RAN,ns) =
        MemoryMergeRAN(b_RAN,ns)

  within ( ( ( ( mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( ( lset.sv_sec.(RAN.0) ->
    ( ( let X = lget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    lget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    lget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( tick ->
    lset.sv_sec.(RAN.(valueRAN(v_sv_sec) + 1) % 3) ->
    lget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ((valueRAN(v_sv_sec) == 0) &
        inc -> SKIP [] (valueRAN(v_sv_sec) != 0) &
        SKIP)
      [] time ->
    minsReq ->
    ans?t_sv_min ->
    mset.sv_min.(RAN.t_sv_min) ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    out.(valueRAN(v_sv_min),valueRAN(v_sv_sec)) -> SKIP);
      X ) within X );
      lterminate -> SKIP )
      [| MEML |]
      MemoryMerge({(sv_min,v_sv_min),(sv_sec,v_sv_sec)},<sv_sec>) )\MEML )
      [| Sync |]
      ( ( lset.sv_min.(RAN.0) ->
    ( ( let X = lget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    ( ( inc ->
    lset.sv_min.(RAN.(valueRAN(v_sv_min) + 1) % 3) -> SKIP
      [] minsReq ->
    ans.valueRAN(v_sv_min) -> SKIP);
      X ) within X );
      lterminate -> SKIP )
      [| MEML |]
      MemoryMerge({(sv_min,v_sv_min),(sv_sec,v_sv_sec)},<sv_min>) )\MEML ) )\Sync );
      terminate -> SKIP )
      [| MEMI |] Memory(b_RAN))\MEMI )
```

```
AChrono(b_RAN) =
  let
     MemoryRANVar(n,b_RAN) =
        ( ( mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
      [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv)))
      [] terminate -> SKIP)
     MemoryRAN(b_RAN) =
        ( [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN) )
     Memory(b_RAN) = MemoryRAN(b_RAN)
     MemoryMergeRANVar(n,b_RAN,ns) =
        ( ( lget.n.apply(b_RAN,n) -> MemoryMergeRANVar(n,b_RAN,ns)
```

```
      [] lset.n?nv:typeRAN(n) -> MemoryMergeRANVar(n,over(b_RAN,n,nv),ns))
      [] lterminate ->
    (  ; bd : <b_RAN> @   ; n : <y | y <- ns,member(y,dom(bd))> @  mset.n.apply(bd,n) -> SKIP ))
     MemoryMergeRAN(b_RAN,ns) =
        ( [| {| lterminate |} |] n : dom(b_RAN) @ MemoryMergeRANVar(n,b_RAN,ns) )
     MemoryMerge(b_RAN,ns) = MemoryMergeRAN(b_RAN,ns)

  within ( ( ( mset.sv_sec.(RAN.0) ->
    mset.sv_min.(RAN.0) ->
    ( let X = mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( tick ->
    mset.sv_sec.(RAN.(valueRAN(v_sv_sec) + 1) % 3) ->
    mset.sv_min.(RAN.valueRAN(v_sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ((valueRAN(v_sv_sec) == 0) &
         mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    mset.sv_min.(RAN.(valueRAN(v_sv_min) + 1) % 3) ->
    mset.sv_sec.(RAN.valueRAN(v_sv_sec)) -> SKIP [] (valueRAN(v_sv_sec) != 0) &
         SKIP)
      [] time ->
    out.(valueRAN(v_sv_min),valueRAN(v_sv_sec)) -> SKIP);
      X ) within X );
      terminate -> SKIP )
      [| MEMI |] Memory(b_RAN))\MEMI )
```

H.4 *WakeUp*

Here we introduce a variant of the *Chronometer* model presented above. The *WakeUp* has the feature of triggering an alarm in case a specific time is set, in our example, when $min = 1$.

$$
\begin{array}{l}
ALARM ::= ON \mid OFF \\
\textbf{channel } snooze, radioOn \\
\textbf{process } \ WakeUp \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state } WState \mathrel{\widehat=} [sec, min : RANGE;\ buzz : ALARM] \\
\quad WInit \mathrel{\widehat=} (sec, min, buzz := 0, 0, OFF) \\
\quad WIncSec \mathrel{\widehat=} (sec, min := (sec + 1) \bmod 3, min) \\
\quad WIncMin \mathrel{\widehat=} (min, sec := (min + 1) \bmod 3, sec) \\
\quad WRun \mathrel{\widehat=} \left(\left(\begin{array}{l} tick \rightarrow WIncSec; \\ \left(\begin{array}{l} \big(sec = 0\big) \mathbin{\&} WIncMin \\ \Box\big(sec \neq 0\big) \mathbin{\&} \mathrm{Skip} \\ \Box\big(min = 1\big) \mathbin{\&} radioOn \rightarrow (buzz := ON) \\ \Box(time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip}) \\ \Box(snooze \rightarrow (buzz := OFF)) \end{array} \right) \end{array} \right) \setminus \{\!| tick |\!\} \right) \\
\quad \bullet\ (WInit\ ;\ \ (\mu X \bullet (WRun\ ;\ \ X))) \\
\textbf{end}
\end{array}
$$

The *WakeUp* model may deadlock. When testing with FDR, we observed that the signal *tick* was the source of the problem. Therefore, the *WakeUpOK* can either tick and update the time, or turn on

the alarm, output the current *time*, or set the alarm to *snooze*.

$$
\begin{array}{l}
\textbf{process}\ \ WakeUpOK \mathrel{\widehat=} \\
\textbf{begin} \\
\quad \textbf{state}\ WState \mathrel{\widehat=} [sec, min : RANGE;\ buzz : ALARM] \\
\quad WInit \mathrel{\widehat=} (sec, min, buzz := 0, 0, OFF) \\
\quad WIncSec \mathrel{\widehat=} (sec, min := (sec + 1) \bmod 3, min) \\
\quad WIncMin \mathrel{\widehat=} (min, sec := (min + 1) \bmod 3, sec) \\
\quad WRun \mathrel{\widehat=} \left(\left(\begin{array}{l} \left(\begin{array}{l} tick \rightarrow WIncSec; \\ \left(\begin{array}{l} (sec = 0) \mathbin{\&} WIncMin \\ \Box (sec \neq 0) \mathbin{\&} \mathrm{Skip} \end{array} \right) \end{array} \right) \\ \Box (min = 1) \mathbin{\&} radioOn \rightarrow (buzz := ON) \\ \Box (time \rightarrow out!(min, sec) \rightarrow \mathrm{Skip}) \\ \Box (snooze \rightarrow (buzz := OFF)) \end{array} \right) \setminus \{\!|\ tick\ |\!\} \right) \\
\quad \bullet\ (WInit\ ;\ \ (\mu X \bullet (WRun\ ;\ \ X))) \\
\textbf{end}
\end{array}
$$

H.4.1 *WakeUp* using *Circus2CSP*

```
1  RANGE = {0..5}
   datatype ALARM = ON | OFF
3
   channel snooze, radioOn
5  channel tick, time
   channel out : (RANGE,RANGE)
```

```
   --------------------------------
2  -- The universe of values
    datatype UNIVERSE = RAN.RANGE | ALA.ALARM
4  --Conversions
   valueRAN(RAN.v) = v
6  valueALA(ALA.v) = v

8  typeRAN(x) = U_RAN
   typeALA(x) = U_ALA
10
   tagRAN(x) = RAN
12 tagALA(x) = ALA

14 -- subtypes of UNIVERSE for RAN
   subtype U_RAN = RAN.RANGE
16
   -- subtypes of UNIVERSE for ALA
18 subtype U_ALA = ALA.ALARM

20 -- definition of NAME for the entire spec
   datatype NAME = sv_sec | sv_min | sv_buzz
22
   -- Subtype definition for RAN
24 b_RAN1 = {(sv_sec, RAN.0),(sv_min, RAN.0)}
   subtype NAME_RAN = sv_sec | sv_min
26 NAMES_VALUES_RAN = seq({seq({(n,v) | v <- typeRAN(n)}) | n <- NAME_RAN})

28 -- Subtype definition for ALA
   b_ALA1 = {(sv_buzz, ALA.ON)}
30 subtype NAME_ALA = sv_buzz
   NAMES_VALUES_ALA = seq({seq({(n,v) | v <- typeALA(n)}) | n <- NAME_ALA})
32
   -- Bindings definitions for RAN
34 BINDINGS_RAN = {set(b) | b <- set(distCartProd(NAMES_VALUES_RAN))}

36 -- Bindings definitions for ALA
   BINDINGS_ALA = {set(b) | b <- set(distCartProd(NAMES_VALUES_ALA))}
```

```
1  --------------------------------
    -- mget, mset and terminate --
3  --------------------------------
   channel mget, mset : NAME.UNIVERSE
5  channel terminate
```

```
--------------------------------
 -- MEMI --
--------------------------------
MEMI = {| mset,mget,terminate |}
channel lget, lset : NAME.UNIVERSE
channel lterminate
MEML = {| lset,lget,lterminate |}
```

```
WakeUp(b_RAN,b_ALA) =
  let
     MemoryRANVar(n,b_RAN) =
        ( ( mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
      [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv)))
      [] terminate -> SKIP)
     MemoryALAVar(n,b_ALA) =
        ( ( mget.n.apply(b_ALA,n) -> MemoryALAVar(n,b_ALA)
      [] mset.n?nv:typeALA(n) -> MemoryALAVar(n,over(b_ALA,n,nv)))
      [] terminate -> SKIP)
     MemoryRAN(b_RAN) = ( [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN) )
     MemoryALA(b_ALA) = ( [| {| terminate |} |] n : dom(b_ALA) @ MemoryALAVar(n,b_ALA) )
     Memory(b_RAN,b_ALA) = ( MemoryALA(b_ALA) [| {| terminate |} |] MemoryRAN(b_RAN) )
  within ( ( ( mset.sv_sec.(RAN.0) ->
    mset.sv_min.(RAN.0) ->
    mset.sv_buzz.(ALA.OFF) ->
    ( let X = tick ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    mset.sv_sec.(RAN.(valueRAN(v_sv_sec) + 1) % 3) ->
    mset.sv_min.(RAN.valueRAN(v_sv_min)) ->
    mget.sv_buzz?v_sv_buzz:(typeALA(sv_buzz)) ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( ( ( ( (valueRAN(v_sv_sec) == 0) & mset.sv_min.(RAN.(valueRAN(v_sv_min) + 1) % 3) ->
    mset.sv_sec.(RAN.valueRAN(v_sv_sec)) -> SKIP )
      [] ( (valueRAN(v_sv_sec) != 0) & SKIP ))
      [] ( (valueRAN(v_sv_min) == 1) & radioOn -> mset.sv_buzz.(ALA.ON) -> SKIP ))
      [] time -> out.(valueRAN(v_sv_min),valueRAN(v_sv_sec)) -> SKIP)
      [] snooze -> mset.sv_buzz.(ALA.OFF) -> SKIP);
      X ) within X );
      terminate -> SKIP )
      [| MEMI |] Memory(b_RAN,b_ALA))\MEMI )
```

```
WakeUpOK(b_RAN,b_ALA) =
  let
     MemoryRANVar(n,b_RAN) =
        ( ( mget.n.apply(b_RAN,n) -> MemoryRANVar(n,b_RAN)
      [] mset.n?nv:typeRAN(n) -> MemoryRANVar(n,over(b_RAN,n,nv)))
      [] terminate -> SKIP)
     MemoryALAVar(n,b_ALA) =
        ( ( mget.n.apply(b_ALA,n) -> MemoryALAVar(n,b_ALA)
      [] mset.n?nv:typeALA(n) -> MemoryALAVar(n,over(b_ALA,n,nv)))
      [] terminate -> SKIP)
     MemoryRAN(b_RAN) = ( [| {| terminate |} |] n : dom(b_RAN) @ MemoryRANVar(n,b_RAN) )
     MemoryALA(b_ALA) = ( [| {| terminate |} |] n : dom(b_ALA) @ MemoryALAVar(n,b_ALA) )
     Memory(b_RAN,b_ALA) = ( MemoryALA(b_ALA) [| {| terminate |} |] MemoryRAN(b_RAN) )
  within ( ( ( mset.sv_sec.(RAN.0) ->
    mset.sv_min.(RAN.0) ->
    mset.sv_buzz.(ALA.OFF) ->
    ( let X = mget.sv_buzz?v_sv_buzz:(typeALA(sv_buzz)) ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( ( ( tick ->
    mset.sv_sec.(RAN.(valueRAN(v_sv_sec) + 1) % 3) ->
    mset.sv_min.(RAN.valueRAN(v_sv_min)) ->
    mget.sv_min?v_sv_min:(typeRAN(sv_min)) ->
    mget.sv_sec?v_sv_sec:(typeRAN(sv_sec)) ->
    ( ( (valueRAN(v_sv_sec) == 0) &
          mset.sv_min.(RAN.(valueRAN(v_sv_min) + 1) % 3) ->
          mset.sv_sec.(RAN.valueRAN(v_sv_sec)) -> SKIP )
      [] ( (valueRAN(v_sv_sec) != 0) & SKIP ))
      [] ( (valueRAN(v_sv_min) == 1) & radioOn -> mset.sv_buzz.(ALA.ON) -> SKIP ))
      [] time -> out.(valueRAN(v_sv_min),valueRAN(v_sv_sec)) -> SKIP)
      [] snooze -> mset.sv_buzz.(ALA.OFF) -> SKIP);
      X ) within X );
      terminate -> SKIP )
      [| MEMI |] Memory(b_RAN,b_ALA))\MEMI )
```

```
-- Experiments in FDR

assert WakeUp(b_RAN1, b_ALA1) :[deterministic [FD]] -- Failed
```

```
assert WakeUpOK(b_RAN1, b_ALA1) :[deterministic [FD]] -- Passed

assert WakeUp(b_RAN1, b_ALA1) :[divergence free [FD]] -- Passed
assert WakeUpOK(b_RAN1, b_ALA1) :[divergence free [FD]] -- Passed

assert WakeUp(b_RAN1, b_ALA1) :[deadlock free [FD]] -- Passed
assert WakeUpOK(b_RAN1, b_ALA1) :[deadlock free [FD]] -- Passed

assert WakeUp(b_RAN1, b_ALA1) [FD= WakeUpOK(b_RAN1, b_ALA1) -- Failed
assert WakeUpOK(b_RAN1, b_ALA1) [FD= WakeUp(b_RAN1, b_ALA1) -- Failed

HWakeUp = WakeUp(b_RAN1,b_ALA1) \ {|tick|}
HWakeUpOK = WakeUpOK(b_RAN1,b_ALA1) \ {|tick|}

assert HWakeUp [FD= HWakeUpOK  -- Passed
assert HWakeUpOK [FD= HWakeUp  -- Passed
```

In memoriam

My dear mother Vilma Maria de Oliveira Silva

☆ *18/01/1954* - ✞ *04/04/2017*