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 CSPM .................................... 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 Rewriting Circus state-rich processes with $\Omega$ 38
   4.1 Upgrading the Memory Model ................................... 41
      4.1.1 Limitation 1: Z types vs. CSPM types .................... 41
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 CSPM (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 CTOCP. 34

3.5 Experiment results derived from Table 3.1 - D241InvPre, D241Pre, CTOC and CTOCP. 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 FDR . . . . . . . . . . . . . 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 adopted 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\textsubscript{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\textsubscript{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\textsubscript{M}:
  
  Implementation of a tool based on the work of Oliveira et al. [132] where one is able to translate Circus models written in \texttt{BLEX} into CSP\textsubscript{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\textsubscript{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\textsubscript{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\textsubscript{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 ≤ 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 CSPM, in order to allow the community to model-check CSPM 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 detail 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].

11
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 \(CSP \parallel B\) 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 possible. 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 \( \text{CSP}_M \) does not have a notion of variables for a state as in \( Z, \text{Circus} \) or even B-Method, we have to somehow capture them in order to obtain a \( \text{CSP}_M \) model as similar as possible to the original \text{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 \text{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 \( \text{CSP}_M \) 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 \( \text{CSP}_M \) 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 \( w : [\text{pre}, \text{post}] \),
and it can be satisfied by an implementation which, given an initial state that satisfies \textit{pre}, the implementation can be executed resulting in a final state updating the variables in the frame \textit{w}, that satisfies \textit{post}.

Moreover, the refinement calculus also defines a relation between the refinement steps, known as \textit{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 \textit{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 \textit{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 \textit{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 \textit{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 \textit{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 mechanismisation of the Unifying Theories of Programming (UTP) into Isabelle/HOL, supporting Circus [58]. Furthermore, Felianchi\ 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 mechanismisation 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:

\[
\text{channel } c : T
\]

For both CSP and Circus, we usually define a specification comprising a set of paragraphs in the form \(N(Expr^+) \equiv A\) 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:

\[ \text{LocVars} \triangleq [v_0 : T_x; \ldots; v_n : T_x | \text{inv}(v_0, \ldots, v_n)] \]

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

\[
\text{process } P \triangleq \\
\begin{array}{l}
\text{begin} \\
\text{state } \text{State} \triangleq \text{LocVars} \\
\text{P}\text{.Actions} \triangleq \Delta \text{State} \ldots \\
\bullet \text{var } l_0 : U_0; \ldots; l_m : U_m \bullet \text{MA}(v_0, \ldots, v_n, l_0, \ldots, l_m) \\
\text{end}
\end{array}
\]

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 \( \text{inv}(v_0, \ldots, v_n) \) regarding the state variables. Moreover, it may have a list of Circus actions \( \text{P}\text{.Actions} \) 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 \( \text{MA} \).

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 (\( \oplus \)), alphabetised parallel (\( \llbracket \ldots \rrbracket \)), interleaving (\( || \)), iterated versions of the above (e.g., \( \sqcap_{e \in E} \ldots \)), and hiding (\( \\backslash \)).

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 (\( \langle \cdot \rangle \)), and recursion (\( \mu \)). 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 \( P \llbracket ns_1 | cs | ns_2 \rrbracket Q \), 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

\[ \text{var } x : T \bullet A \]

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.

\[ (\text{vres } x : T \bullet A)(y) \]
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 $\text{RANGE}$ 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 $\text{tick}$ every second; when the $\text{time}$ is requested, the channel $\text{out}$ outputs the minutes and seconds as a pair; $\text{radioOn}$ turns on the radio when the alarm is set; and $\text{snooze}$ mutes the alarm.

The events $\text{snooze}$ and $\text{radioOn}$ are part of the added alarm features.

\[
\begin{align*}
\text{RANGE} &\equiv 0 .. 2 \\
\text{ALARM} &::= \text{ON} \mid \text{OFF} \\
\text{channel} \text{tick}, \text{time}, \text{snooze}, \text{radioOn} \\
\text{channel} \text{out} &::= \{ \text{min}, \text{sec} : \text{RANGE} \cdot (\text{min}, \text{sec}) \}
\end{align*}
\]

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

\[
\begin{align*}
\text{process} \text{WakeUp} &\triangleq \begin{array}{c}
\text{begin} \\
\text{state} \text{WState} &\triangleq [\text{sec}, \text{min} : \text{RANGE}; \text{buzz} : \text{ALARM}]
\end{array}
\end{align*}
\]

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, $\text{AInit}$, sets both $\text{sec}$ and $\text{min}$ to zero, as well as $\text{buzz}$ to $\text{OFF}$. Moreover, $\text{IncSec}$ and $\text{IncMin}$ increment $\text{sec}$ and $\text{min}$ respectively.

\[
\begin{align*}
\text{AInit} &\triangleq [AState' | \text{sec}' = 0; \text{min}' = 0] \\
\text{IncSec} &\triangleq [\Delta AState | \text{sec}' = (\text{sec} + 1) \mod 60] \\
\text{IncMin} &\triangleq [\Delta AState | \text{min}' = (\text{min} + 1) \mod 60]
\end{align*}
\]

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

¹This work is further detailed in Section 3.1.
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 hiding 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\textsubscript{M} translation strategy developed for the EU COMPASS project and described in deliverable D24.1 \cite[Section 5]{132}. It specifies the translation of a state-rich Circus specification into a defined subset of Circus by describing functions \((\Omega_P, \Omega_A, \Upsilon)\) that transform Circus syntax. The translation process starts with a state-rich Circus\textsubscript{SR} process \(P\), which, applying \(\Omega_P\) (\(P\) for Circus processes), is then transformed into a state-poor Circus\textsubscript{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\textsubscript{M} (derived from \cite[Fig.7, p77]{132})](image)

The Omega functions, \(\Omega_P\) and \(\Omega_A\), 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 \texttt{mget} and \texttt{mset} channels to provide read and write operations. The memory parameters required are determined by \(\Omega_P\) when applied to the top-level or ‘main’ Circus process.

In this transformation procedure, the Circus refinement laws along with the \(\Omega_A\) (\(A\) 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 \(\Omega\) functions result in models that belong to Circus\textsubscript{CSP}, the CSP subset of Circus containing only “state-poor” divergence-free Circus processes, ready to be translated into CSP\textsubscript{M} by \(\Upsilon\). Finally, the \(\Upsilon\) function takes the results of the \(\Omega\) functions, now corresponding to standard CSP, and renders it in machine-readable CSP\textsubscript{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 hiding.

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

\(^1\)The remainder of the \(\Omega\) functions are defined in Section 5.3 of the Deliverable report D24.1\cite{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 highlight 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 nondeterministically, and therefore, we are no longer able to prove the equivalence between $P_{SR}$ and $\Omega(P_{SR})$, 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 $\Omega$ 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 $\Omega_{\text{Step}_n} \sqsubseteq \Omega_{\text{Step}_{n+1}}$, we also expect to prove that the whole refinement is also correct:

$$P_{SR} \sqsubseteq \ldots \sqsubseteq \Omega_{\text{Step}_n} \sqsubseteq \Omega_{\text{Step}_{n+1}} \sqsubseteq \ldots \sqsubseteq P_{SL}$$

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 $\text{UNIVERSE}$ comprising any type defined in the specification. Moreover, the names of every state component and local variable are defined as components of $\text{NAME}$.

\[
\text{UNIVERSE} \\
\text{NAME} ::= sv_{v_1} | sv_{v_2} | \ldots | sv_{v_n} | lv_{l_1} | \ldots | lv_{l_n}
\]

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, \textit{BINDING}, which maps all the names, \textit{NAME}, into the \textit{UNIVERSE} type.

\[
\text{BINDING} == \text{NAME} \rightarrow \text{UNIVERSE}
\]

\[
\delta == \{sv_{v_1} \mapsto T_1, sv_{v_2} \mapsto T_2, \ldots, sv_{v_n} \mapsto T_3, \ldots, lv_{l_n} \mapsto T_m\}
\]

As a result of applying the \(\Omega\) functions, the state of a \textit{Circus} process is replaced by a \textit{Memory} action which manages the values, offering \textit{gets} and \textit{sets}, for the values of the state components of such a process, when translated into \textit{CSP}_M.

\[
\text{Memory} \equiv \text{vres} b : \text{BINDING} \bullet
\]

\[
\quad (\Box n : \text{dom} b \bullet \text{mget}.n!b(n) \rightarrow \text{Memory}(b))
\]

\[
\quad \Box (\Box n : \text{dom} b \bullet \text{mset}.n?nv : (\text{nv} \in \delta(n)) \rightarrow \text{Memory}(b \oplus \{n \mapsto \text{nv}\}))
\]

\[
\quad \Box \text{terminate} \rightarrow \text{Skip}
\]

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

\[
\text{channelset } \text{MEM}_I == \{\text{mget, mset, terminate}\}
\]

The final specification puts the original process after \(\Omega\)-translation in parallel with the memory model, synchronising on the \textit{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:

\[
\cap b : \text{BINDING} \bullet \left(\Omega_A(P); \text{terminate} \rightarrow \text{Skip}\right) \setminus \text{MEM}_I
\]

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 \textit{Circus} construct, in order to provide a translation into \textit{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 \textit{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 \textit{Circus} constructs. When he moved to the implementation of complex actions in \textit{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 \( \text{LATEX} \) 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 \( \text{LATEX} \) 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 \( \text{CSP}_M \) 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 \( \text{LATEX} \). 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 \( \text{FactX} \) 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 provided is already type checked with existing tools [109]. After parsing the \( \text{LaTeX} \) 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.

\[
ZPara ::= \begin{array}{l}
\text{channel} \ CDecl \\
\text{channelset} \ N \ = \ CExp \\
\text{process} \ P \ \triangleq \ \text{ProcDef}
\end{array}
\]

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

```haskell
1 data ZPara =
2 | ...  
3 | CircChannel {CDecl}  
4 | CircChanSet {ZName} {CExp}  
5 | Process {ProcDecl}  
6 | ...
```

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.
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 $\Omega$ 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 \( \text{Just } y \) or an empty value, (denoted by \( \text{Nothing} \)). We evaluate such a type in order to assert whether 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 \( \text{Done} \), that contains the matched instance of that left-hand side (\( \text{orig} \)), along with the corresponding right-hand side (\( \text{refined} \)), plus a list of side-conditions that need to be satisfied (\( \text{provisos} \)). However, when the refinement is not applied to that specification, the calculator returns \( \text{None} \). 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.

We illustrate our approach with the example of a \textit{Circus} refinement law, \textbf{guard combination}, L. 15\textsuperscript{3}, which allows merging bindings within the context of the action \( A \).

\[
(g_1) \& ((g_2) \& A) = (g_1 \land g_2) \& A
\]

The above refinement law is implemented in Haskell as the following function \( \text{crl\_guardComb} \) that uses pattern matching to check for abstract syntax that matches the left-hand side of the law above, and returns the refinement in \( \text{refined} \). 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 \( \text{None} \).

\begin{verbatim}
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
\end{verbatim}

The second example we use here to illustrate our implementation of the refinement laws is the \textbf{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: \( \text{initials}(A_2) \subseteq cs \).

\[
(((g) \& A_1) \sqcap cs \sqcap ns_1 \sqcup A_2) = (g) \& (A_1 \sqcap ns_1 \sqcup cs \sqcup ns_2 \sqcup A_2)
\]

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 \( \text{None} \), meaning that no refinement was produced. In either two first clauses, where \( \text{Done} \) is returned, the proviso field contains the pending proof obligation regarding the set of initials of the action in parallel with the guarded action.

\begin{verbatim}
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 (ZMember (ZTuple [ZSetDisplay (initials a2),
                            ZSetDisplay (zname_to_zexpr cs)]) [ZVar ("\"\subset\\\"",[],[])]) [ZVar ("\\\subset\\\"",[1,1])])))}
crl_guardParDist _ = None
\end{verbatim}

\textsuperscript{3}The compilation of the \textit{Circus} refinement laws mentioned in this thesis can be found in the Appendix D.
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:

process $P \equiv$
begin
state $S$
$$L(x : T)$$
• ($var x : T$ • $MA$)
end

process $P \equiv$
begin
state $S \land [x : T]$
$$L(\_\_)$$
• $MA$
end

as well as for those without a preexisting state paragraph:

process $P \equiv$
begin
$$L(x : T)$$
• ($var x : T$ • $MA$)
end

process $P \equiv$
begin
$$(state[x : T])$$
$$L(\_\_)$$
• $MA$
end

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 \rightarrow 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
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 \( A_1 \sqcap A_2 \). 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 \( A_1 \equiv c_1 \Rightarrow \text{Skip} \) and \( A_2 \equiv c_2 \Rightarrow \text{Skip} \), 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, \( v_1 = 0 \land v_2 = 0 \).

\[
\begin{align*}
(v_1 > 0) & \land (v_2 > 0) \land A_1 \left[ \varnothing \right] \left( \{c_1, c_2\} \right] \varnothing A_2 \\
& = [L. 19 - \text{prov:initials}(A_2) \subseteq \{c_1, c_2\}] \\
(v_1 > 0) & \land (v_2 > 0) \land (A_1 \left[ \varnothing \right] \left( \{c_1, c_2\} \right] \varnothing A_2 ) \\
& = [L. 15] \\
(v_2 > 0 \land v_1 > 0) & \land (A_1 \left[ \varnothing \right] \left( \{c_1, c_2\} \right] \varnothing A_2 )
\end{align*}
\]

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.
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 assignments 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 CSPM 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 CSPM, 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, \( St \equiv [d \mid inv] \). 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:

\[
[D\ St;\ \text{di}?;\ \text{do!}\ |\ p] =_{bC} \alpha \text{d}, \alpha \text{do!} : [\text{inv} \land \exists \text{d'};\ \text{do!} \bullet \text{inv'} \land p, \text{inv'} \land p]
\]

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.

\[
[D\ St;\ \text{di}?;\ \text{do!}\ |\ p] = [\text{normalisation}]
\]

\[
[\text{St};\ \text{St'};\ \text{di}?;\ \text{do!}\ |\ \text{inv} \land p] = [\text{129, def B.40}]
\]

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 \(CSP_M\).

\[
\Omega_A \left( \begin{array}{l}
\text{pre}(v_0, \ldots, v_n), \\
\text{post}(v_0, \ldots, v_n, l_0, \ldots, l_n)
\end{array} \right) \Rightarrow
\begin{cases}
\text{mget}\cdot v_0?v_0 \rightarrow \ldots \rightarrow \text{mget}\cdot v_0?v_n \rightarrow \\
\text{mget}\cdot l_0?v_0 \rightarrow \ldots \rightarrow \text{mget}\cdot l_n?v_n \rightarrow
\neg (\text{pre}(v_0, \ldots, v_n, l_0, \ldots, l_n)) \land \text{Chaos}
\end{cases}
\]

\[
\begin{array}{l}
\square (\text{pre}(v_0, \ldots, v_n, l_0, \ldots, l_n)) \land \bigwedge (x_0 : \Gamma(v_0); \ldots; x_n : \Gamma(v_n); \\
\quad \left(\begin{array}{c}
\quad x_{n+1} : \Gamma(l_0); \ldots; x_m : \Gamma(l_m) \\
\quad \text{mset}\cdot v_0!(v_0.0) \rightarrow \ldots \rightarrow \\
\quad \text{mset}\cdot v_0!(v_0.n) \rightarrow \\
\quad \text{mset}\cdot l_0!(v_0.(n + 1)) \rightarrow \ldots \rightarrow \\
\quad \text{mset}\cdot l_m!(v_0.(n + 1)) \rightarrow \\
\quad \text{Skip}
\end{array}\right) \cdot \\
\quad \text{inv} \lor \forall \mathcal{W} = \mathcal{W}
\end{array}
\]

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:

\[
\Delta \text{St}; \text{di?; do!} | c'_1 = e_1 \land \ldots \land c'_n = e_n \land o_1! = e_{n+1} \land o_m! = e_{n+m}
\]

\[
\sqsubseteq \quad \text{assC - provided } \text{inv}[e_1, \ldots, e_n/c_1, \ldots, c_n] \quad \text{and} \quad c'_1, \ldots, c'_n \not\in \text{FreeVars}(e_1, \ldots, e_{n+m})
\]

\[
c_1, \ldots, c_n, o_1!, \ldots, o_m! := e_1, \ldots, e_{n+m}
\]

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

\[
\Delta \text{St} | c'_1 = e_1 \land \ldots \land c'_n = e_n
\]

\[
\sqsubseteq \quad \text{assC - provided } \text{inv}[e_1, \ldots, e_n/c_1, \ldots, c_n] \quad \text{and} \quad c'_1, \ldots, c'_n \not\in \text{FreeVars}(e_1, \ldots, e_n)
\]

\[
c_1, \ldots, c_n := e_1, \ldots, e_n
\]

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 \text{min} and \text{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 \text{CSP}_M, it is safe to restrict the range of \text{min} and \text{sec} to \(0 \ldots 60\), where 60 is an unexpected value in the system.

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

We experimented with the impact of explicitly including invariant and precondition checks in \text{Circus} models and translating them into \text{CSP}_M using the example of the Chronometer, from Section 2.4, with a new process \text{Chrono}. When using the translation rules presented in \cite{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 \text{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 \text{pre} is not satisfied, it behaves such as \text{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.

\[
\text{process } Chrono \triangleq \\
\text{begin} \\
\text{state } AState \triangleq [sec, min : \mathbb{N} | min < 60 \land sec < 60] \\
AInit \triangleq [AState' | sec' = 0; min' = 0] \\
IncSec \triangleq [\Delta AState | sec' = (sec + 1) \mod 60] \\
IncMin \triangleq [\Delta AState | min' = (min + 1) \mod 60] \\
Run \triangleq \\
\left( \begin{array}{c}
tick \rightarrow (IncSec); \\
\square (sec = 0) \land (IncMin) \\
\square (sec \neq 0) \land \text{Skip} \\
\square \text{time} \rightarrow \text{out}!(min, sec) \rightarrow \text{Skip}
\end{array} \right) \\
\bullet AInit ; (\mu X \bullet Run ; X) \\
\text{end}
\]

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\(^4\):

D241 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.

D241Inv Model translated using the approach from [132] including the invariants as a restriction to the bindings set.

D241Pre 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.

D241InvPre Combination of D241Inv and D241Pre.

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).

CTOCP re Extension of CTOC model but with the same translation rules for D241Pre, 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 performed 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..10, the process state is then defined as \([min, sec : 0..10 | min < 10 \land sec < 10]\).

\(^4\)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

<table>
<thead>
<tr>
<th>Values Range</th>
<th>CTOC (states visited)</th>
<th>CTOCPre (states visited)</th>
<th>D241 (states visited)</th>
<th>D241Inv (states visited)</th>
<th>D241InvPre (states visited)</th>
<th>D241Pre (states visited)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..3</td>
<td>0.116</td>
<td>0.116</td>
<td>0.206</td>
<td>0.177</td>
<td>0.173</td>
<td>0.187</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.116</td>
<td>0.206</td>
<td>0.177</td>
<td>0.173</td>
<td>0.187</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.116</td>
<td>0.206</td>
<td>0.177</td>
<td>0.173</td>
<td>0.187</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.116</td>
<td>0.206</td>
<td>0.177</td>
<td>0.173</td>
<td>0.187</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.116</td>
<td>0.206</td>
<td>0.177</td>
<td>0.173</td>
<td>0.187</td>
</tr>
<tr>
<td></td>
<td>0.116</td>
<td>0.116</td>
<td>0.206</td>
<td>0.177</td>
<td>0.173</td>
<td>0.187</td>
</tr>
</tbody>
</table>

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.
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](image)

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 affected by 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.

\[ \text{RANGE} \equiv \{0 .. 59\} \]

\[ \text{channel} ~ \text{tick, time} \]

\[ \text{channel} ~ \text{out} : \text{RANGE} \times \text{RANGE} \]

\[ \text{process} ~ \text{Chrono} \equiv \]

\[ \begin{align*}
\text{begin} \\
\text{state} \ AState &\equiv [\text{sec, min} : \mathbb{N} \mid \text{min} < 60 \land \text{sec} < 60] \\
AInit &\equiv [AState' \mid \text{sec}' = 0; \text{min}' = 0] \\
IncSec &\equiv [\Delta AState \mid \text{sec}' = (\text{sec} + 1) \mod 60] \\
IncMin &\equiv [\Delta AState \mid \text{min}' = (\text{min} + 1) \mod 60] \\
\text{Run} &\equiv \begin{cases} 
\text{tick} \rightarrow (\text{IncSec}); \\
\left( \begin{array}{c}
\text{sec} = 0) \& (\text{IncMin}) \\
\text{sec} \neq 0 \& \text{Skip} \\
time \rightarrow \text{out}!(\text{min}, \text{sec}) \rightarrow \text{Skip}
\end{array} \right)
\end{cases}
\end{align*} \]

\[ \bullet \ AInit ; (\mu X \cdot \text{Run} ; X) \]

\text{end}

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 (\(\Omega\)) functions, into a *Circus* syntax which can be translated
into CSP\(_M\).

\[ \begin{align*}
\text{process} \ AChrono \equiv \begin{align*}
\text{begin} \\
\text{state} \ AState &\equiv [\text{sec, min} : \mathbb{N} \mid \text{min} < 60 \land \text{sec} < 60] \\
\mu X \bullet \\
\left( \begin{array}{c}
\text{tick} \rightarrow (\text{sec} := (\text{sec} + 1) \mod 60); \\
\left( \begin{array}{c}
\text{sec} = 0) \& (\text{min} := (\text{min} + 1) \mod 60) \\
\text{sec} \neq 0 \& \text{Skip} \\
\text{time} \rightarrow \text{out}!(\text{min}, \text{sec}) \rightarrow \text{Skip}
\end{array} \right)
\end{array} \right) ; X
\end{align*} \]

\text{end}

\[ \begin{align*}
\text{process} \ AChrono \equiv \begin{align*}
\text{begin} \\
\text{Memory} &\equiv \ldots \\
\mu X \bullet \\
\left( \begin{array}{c}
\text{mset.\_sv\_sec}.(\text{RAN} \cdot 0) \rightarrow \text{mset.\_sv\_min}.(\text{RAN} \cdot 0) \rightarrow \\
\left( \begin{array}{c}
\text{tick} \rightarrow \text{mset.\_sv\_sec}.(\text{RAN} \cdot (\text{v\_sv\_sec} + 1) \mod 3) \rightarrow \\
\text{mget.\_sv\_min}?.\text{v\_sv\_min} : (\delta_{\text{RAN}}(\text{sv\_min})) \rightarrow \\
\text{mget.\_sv\_sec}?.\text{v\_sv\_sec} : (\delta_{\text{RAN}}(\text{sv\_sec})) \rightarrow \\
\left( \begin{array}{c}
\text{v\_sv\_sec} = 0) \& \\
\text{mset.\_sv\_min}.(\text{RAN} \cdot (\text{v\_sv\_min} + 1) \mod 3) \rightarrow \text{Skip}
\end{array} \right)
\end{array} \right) ; X
\end{array} \right)
\end{align*} \]

\text{end}

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 \(\Omega\)

The translation steps using the \(\Omega\) 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.

\[
\text{process } P \equiv \\
\text{begin} \\
\text{state State} \equiv [v_0 : T_0; \ldots; v_n : T_n | \text{inv}(v_0, \ldots, v_n)] \\
P.\text{Actions} \equiv_{\Delta} \text{P.State} \\
\bullet \text{MA}(v_0, \ldots, v_n, l_0, \ldots, l_m) \\
\text{end} \\
\tag{4.1}
\]

Given the initial shape of the process \(P\), the first iteration is to expand the definitions of all the \(P.\text{Actions}\) into \(\text{MA}\), 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, \(\text{MA}\), 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.

\[
\text{process } P \equiv \\
\text{begin} \\
\text{state State} \equiv [v_0 : T_0; \ldots; v_n : T_n; l_0 : U_0; \ldots; l_m : U_m | \text{inv}(v_0, \ldots, v_n)] \\
\bullet \text{var} l_0 : U_0; \ldots; l_m; U_m \bullet \text{MA}(v_0, \ldots, v_n, l_0, \ldots, l_m) \\
\text{end} \\
\tag{4.2}
\]

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.

\[
\text{process } P \equiv \\
\text{begin} \\
\text{state State} \equiv [v_0 : T_0; \ldots; v_n : T_n; l_0 : U_0; \ldots; l_m : U_m; \text{inv}(v_0, \ldots, v_n)] \\
\bullet \text{A}(v_0, \ldots, v_n, l_0, \ldots, l_m) \\
\text{end} \\
\tag{4.3}
\]

\(^1\)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 \( \text{BINDING} \), as already presented here.

\[
\text{process } P \triangleq \\
\text{begin} \\
\text{state } State \triangleq [b : \text{BINDING} \mid b(v_0) \in T_0 \land \ldots \land \text{inv}(b(v_0), \ldots, b(l_m))] \\
\bullet A(b(v_0), \ldots, b(v_n), b(k), \ldots, b(l_m)) \\
\text{end}
\] (4.4)

Now that the state is redefined concerning the bindings, we then introduce a new \textit{Circus} action, \textit{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 \textit{decl} of a process state can be also be defined as a local variable of the main action, \textit{var decl} \( A \).

Therefore, the declaration of the bindings in \textit{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 \textit{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.

\[
\text{process } P' \triangleq \\
\text{begin} \\
\text{state } State \triangleq [b : \text{BINDING} \mid b(v_0) \in T_0 \land \ldots \land \text{inv}(b(v_0), \ldots, b(l_m))] \\
\text{Memory} \triangleq \\
\text{vres } b : \text{BINDING} \bullet \\
\quad (\Box n : \text{dom } b \bullet mget.n!b(n) \rightarrow \text{Memory}(b)) \\
\quad (\Box n : \text{dom } b \bullet (\text{mset.n?nv : } (nv \in \delta(n)) \rightarrow \text{Memory}(b \oplus n \mapsto nv))) \\
\quad \Box \text{terminate} \rightarrow \text{Skip} \\
\bullet \text{var } b : \text{BINDING} \bullet \\
\quad ((\Omega_A(A); \text{terminate} \rightarrow \text{Skip}) \exists [\varnothing | \text{MEM}_I | \{b\}] \text{Memory}(b) \setminus \text{MEM}_I) \\
\text{end}
\] (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 \textit{Circus} process.

\[
\text{process } P' \triangleq \\
\text{begin} \\
\text{Memory} \triangleq \\
\text{vres } b : \text{BINDING} \bullet \\
\quad (\Box n : \text{dom } b \bullet mget.n!b(n) \rightarrow \text{Memory}(b)) \\
\quad (\Box n : \text{dom } b \bullet (\text{mset.n?nv : } (nv \in \delta(n)) \rightarrow \text{Memory}(b \oplus n \mapsto nv))) \\
\quad \Box \text{terminate} \rightarrow \text{Skip} \\
\bullet \text{var } b : \text{BINDING} \bullet \\
\quad ((\Omega_A(A); \text{terminate} \rightarrow \text{Skip}) \exists [\varnothing | \text{MEM}_I | \{b\}] \text{Memory}(b) \setminus \text{MEM}_I) \\
\text{end}
\] (4.6)

According to the example of the \textit{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 \( \text{BINDING} \). 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* 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 *var x* is equivalent to $\cap \{ \text{var } x := k \mid k \in T \}$. In other words, the initial value of a declared variable *x* is arbitrarily non-deterministic. Thus, this can be rewritten as *var x* : *T* $\equiv \cap$ (*var* $\cap$ *x*: *T* $\Rightarrow$ *Memory*, *b*)

Whilst testing our tool, we identified that by translating the Step 4.7 into *CSP* model, 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 expressive 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_splitting*, 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* model.

process $P' \equiv b : BINDING$
begin
Memory \equiv
vres b : BINDING •
(\square n : dom b • \text{mget.n!b}(n) \rightarrow Memory(b))
\square (\square n : dom b • \text{mset.n?nv} : (nv \in \delta(n)) \rightarrow Memory(b \oplus n \mapsto nv))
\square \text{terminate} \rightarrow \text{Skip}
\bullet \cap b : BINDING •
(\Omega_A(A) \land \text{terminate} \rightarrow \text{Skip})
\{\emptyset | MEM_I \cup \{b\}\}
\text{Memory(b)}
\end

41
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.](image)

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: \( \mathbb{Z} \) 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 explicitly declaring the types involved in a specification environment. As of now, we are not detailing the \(CSP_M\) specifics: 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.

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

\[
MemoryTVar(n, b) \equiv \begin{cases} mget.n.b(n) \rightarrow MemoryTVar(n, b) \\ \Box mset.n?nv : \delta(n) \rightarrow MemoryTVar(n, b \oplus \{n \mapsto nv\}) \\ \Box terminate \rightarrow Skip \end{cases}
\] (4.11)
Remember that $\delta$ 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.

$$\text{Memory}(b_{T_1}, \ldots, b_{T_n}) \equiv MemoryT_{1}(b_{T_1})[\{\text{terminate}\} \ldots \{\text{terminate}\}] MemoryT_{k}(b_{T_k}) \quad (4.12)$$

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

$$\text{process } P' \equiv b_{T_1} : \text{BINDING}_{T_1}, \ldots, b_{T_n} : \text{BINDING}_{T_n}$$

$$\begin{array}{l}
\text{begin} \\
\quad MemoryTVar(n, b) \equiv \\
\quad \{\text{mget}.n.b(n) \rightarrow MemoryTVar(n, b)\} \\
\quad \{\text{mset}.n?nv : \delta(n) \rightarrow MemoryTVar(n, b \oplus n \rightarrow nv)\} \\
\quad \{\text{terminate} \rightarrow \text{Skip}\} \\
\text{end}
\end{array}$$

$$\text{Memory}(b_{T_1}, \ldots, b_{T_n}) \equiv$$

$$\begin{pmatrix}
\text{Memory}_{1}(b_{T_1}) \\
\vdots \\
\text{Memory}_{k}(b_{T_k})
\end{pmatrix}\{\text{terminate}\} [\{\text{terminate}\}] \ldots \{\text{terminate}\} \\
\{\begin{array}{l}
\text{MEM}_{I} | \{b_{T_1}, \ldots, b_{T_n}\}\} \\
\text{MEM}_{I} \end{array}\} \{\text{MEM}_{I}\} \text{end}$$

4.2 Rewriting Circus Actions with $\Omega$

We implemented the $\Omega$ functions that transform state-rich into state-poor Circus processes. The implementation of the Omega functions 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.

$$\Omega_A(c \rightarrow A) \equiv c \rightarrow \Omega_A(A)$$

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 $a$
is the subsequent action. In this case, we use the pattern \((\text{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 \(\Omega_A\) for such pattern recurses over \(a\).

\[
\omega_{\text{CActions}} :: \text{CAction} \rightarrow \text{CAction}
\]

\[
\omega_{\text{CActions}} \left( \text{CSPCommAction} \left( \text{ChanComm } c \[\] \right) a \right) = \left( \text{CSPCommAction} \left( \text{ChanComm } c \[\] \right) (\omega_{\text{CActions}} a) \right)
\]

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 \(\text{mget}\) events, retrieving the values of the related state variables to the expression \(e\).

\[
\Omega_A(\text{c.e}(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \equiv mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \text{c.e}(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow \Omega_A(A)
\]

where \(\text{FV}(e) = (v_0, \ldots, v_n, l_0, \ldots, l_m)\)

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 \(\text{getChanDotExpVar}\). 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 \(\text{mget}\) event is created in order to retrieve the value of that state variable. In \text{Circus2CSP}, we use the auxiliary function \(\text{make_get_com}\) which creates the \(\text{mget}\) 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 \(\text{rename_vars_CAction}\), so they now refer to the \(v_\_\) prefixed variables, obtained using \(\text{mget}\), rather than accessing the state variables directly.

\[
\omega_{\text{CAction}} \left( \text{CSPCommAction} \left( \text{ChanComm } c \ e \right) a \right) = \text{case } fvars \text{ of }
\]

\[
\begin{aligned}
\left[\right] & \rightarrow \left( \text{CSPCommAction} \left( \text{ChanComm } c \ e \right) \left( \omega_{\text{CAction}} a \right) \right) \\
\_ & \rightarrow \text{make_get_com } fvars \left( \text{rename_vars_CAction} \left( \text{CSPCommAction} \left( \text{ChanComm } c \ e \right) \left( \omega_{\prime\text{CAction}} a \right) \right) \right) \\
\end{aligned}
\]

where \(fe = \text{getChanDotExpVar } e\)

\[
\text{fvars = (remdups \ (\text{concat} \ (\text{map get\_ZVar\_st} \ (\text{concat} \ (\text{map varset\_to\_zvars} \ (\text{map free\_var\_ZExpr } fe)))))))}
\]

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 \(\text{mget}\) prefixed action should be performed right before the scope of the operator, so those values can then be used in both sides. Mainly, the \(\Omega_A\) set of action behaves just like \(\Omega_A\), however, does not produce any \(\text{mget}\) or \(\text{mset}\) before the construct.

\[
\Omega_A(A_1 \triangleright A_2) \equiv \begin{pmatrix}
mget.v_0?vv_0 \rightarrow \ldots \rightarrow mget.v_n?vv_n \rightarrow mget.l_0?vl_0 \rightarrow \ldots \rightarrow mget.l_m?vl_m \rightarrow \\
\Omega_A(A_1) \triangleright \Omega_A(A_2)
\end{pmatrix}
\]

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 \(\text{mget}\) events. Finally, the \(\Omega_A\) functions are used for both sides of the external choice, meaning that the

---

3The source code of Circus2CSP is available online [76].
values obtained with \textit{mget} are propagated to each side of the choice, and no further \textit{mget} events should happen until required, as defined by the translation rules.

\begin{verbatim}
omega_CAction (CSPExtChoice ca cb) = make_get_com fvvars (CSPExtChoice (omega_prime_CAction ca) (omega_prime_CAction cb))
where
fvvars = remdups $ concat $ (map get_ZVar_st $ varset_to_zvars $
  free_var_CAction (CSPExtChoice ca cb))
\end{verbatim}

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 $\Omega_A$ function is called again for such expanded form.

\begin{align*}
\Omega_A(\langle x : (v_1, ..., v_n) \rangle \cdot A(x)) & \equiv \Omega_A(A(v_1); \ldots; A(v_n)) \\
\Omega_A(\Box x : (v_1, ..., v_n) \cdot A(x)) & \equiv \Omega_A(A(v_1) \Box \ldots \Box A(v_n))
\end{align*}

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 \textit{AInit} action of the \textit{WakeUp} Circus process from Section 2.4.2, where the assignment operator is used in order to set the initial values for \textit{sec} and \textit{min} as 0, and \textit{buzz} as \textit{OFF}.

\begin{align*}
\Omega_A(sv_{sec}, sv_{min}, sv_{buzz} := 0, 0, OFF) \equiv \\
\begin{cases}
  mset. sv_{sec}.0 \to \\
  mset. sv_{min}.0 \to Skip \\
  mset. sv_{buzz}.OFF \to Skip
\end{cases}
\end{align*}

We can observe that no \textit{mget} event occurred in \textit{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 \textit{mget} and \textit{mset} events is illustrated below. Let’s translate now the action \textit{IncSec} from the same example of Section 2.4.2:

\begin{align*}
\Omega_A(sec, min := (sec + 1) \mod 3, min) \equiv \\
\begin{cases}
  mget. sv_{sec}?v_{sv_{sec}} : \delta(sv_{sec}) \to \\
  mget. sv_{min}?v_{sv_{min}} : \delta(sv_{min}) \to \\
  mset. sv_{sec}.(NAT: ((v_{sv_{sec}} + 1) \mod 3)) \to \\
  mset. sv_{min}.v_{sv_{min}} \to Skip
\end{cases}
\end{align*}

Basically, the result transformation is a sequence of \textit{mgets} and \textit{msets} from the \textit{Memory} model. We first get, \textit{mget}, all the state variable and local variable values used in the left-hand side of the assignment, and then, using \textit{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 $A_1 \parallel ns_1 | cs | ns_2 \parallel A_2$. 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: \textit{MemoryMerge} and \textit{Merge}. The former behaves like \textit{Memory} using a copy of the bindings, except in the case of a \textit{terminate} signal, when it propagates two other signals \textit{mleft} and \textit{mright} to the later, which will write the final values to \textit{Memory}. 

46
resulting in several actions, where we first split it into several copies, one for each binding type, synchronising on \text{terminate}.

We follow the idea of the distributed memory system and therefore, we define a distributed MemoryMerge according to the state partition.

\[
\Omega_A(A_1 \parallel ns_1 | cs | ns_2 \parallel A_2) \equiv \\
mget(v_0?v_0 \rightarrow \ldots \rightarrow mget(v_n?v_n \rightarrow \\
mget(l_0?v_0 \rightarrow \ldots \rightarrow mget(l_m?v_m \rightarrow \\
\left( \begin{array}{c}
(\Omega'_A(A_1); \text{terminate} \rightarrow \text{Skip}) \\
(\varnothing | \text{MEM}_L | \varnothing) \\
\text{MemoryMerge}(\{v_0 \Rightarrow v_0, \ldots \Rightarrow \text{LEFT})
\end{array} \right) \setminus \text{MEM}_L
\right)
\]

\[
\left( \begin{array}{c}
(\Omega'_A(A_2); \text{terminate} \rightarrow \text{Skip}) \\
(\varnothing | \text{MEM}_L | \varnothing) \\
\text{MemoryMerge}(\{v_0 \Rightarrow v_0, \ldots \Rightarrow \text{RIGHT})
\end{array} \right) \setminus \text{MEM}_L
\right)
\]

where

\[
\text{Merge} \equiv \\
\left( \begin{array}{c}
mleft(l \rightarrow n : ns_1 \bullet \text{mset.n!l}(n) \rightarrow \text{Skip}) \\
\varnothing (mright(l \rightarrow n : ns_2 \bullet \text{mset.n!l}(n) \rightarrow \text{Skip})
\end{array} \right)
\]

\[
\text{MemoryMerge} \equiv \\
\left( \begin{array}{c}
\text{vres} b : \text{BINDING}; s : \text{SIDE} \bullet \\
\left( \begin{array}{c}
\text{dom} b \bullet \text{mget.n!l}(n) \rightarrow \\
\text{MemoryMerge}(b, s)
\end{array} \right)
\end{array} \right)
\]

In our model, we redesigned the translation rule for parallel actions. We isolate completely the local copies of initial values of the bindings with \(\Gamma_A\) functions instead of \(\Omega_A\), which uses local gets and sets, namely \text{get} and \text{set}, as an intermediate state. It will propagate the local state values from \text{MemoryMerge} to Memory should a \text{terminate} signal occurs, meaning that the parallelism has finished. Hence, the new \(\Omega_A\) translation rule for parallel actions is illustrated as follows.

\[
\Omega_A(A_1 \parallel ns_1 | cs | ns_2 \parallel A_2) \equiv \\
mget(v_0?v_0 \rightarrow \ldots \rightarrow mget(v_n?v_n \rightarrow \\
mget(l_0?v_0 \rightarrow \ldots \rightarrow mget(l_m?v_m \rightarrow \\
\left( \begin{array}{c}
(\Gamma'_A(A_1); \text{terminate} \rightarrow \text{Skip}) \\
(\varnothing | \text{MEM}_L | \varnothing) \\
\text{MemoryMerge}(b_{\text{TYP}_1}, \ldots, b_{\text{TYP}_n}, ns_1) \setminus \text{MEM}_L
\end{array} \right)
\right)
\]

\[
\left( \begin{array}{c}
(\Gamma'_A(A_2); \text{terminate} \rightarrow \text{Skip}) \\
(\varnothing | \text{MEM}_L | \varnothing) \\
\text{MemoryMerge}(b_{\text{TYP}_1}, \ldots, b_{\text{TYP}_n}, ns_2) \setminus \text{MEM}_L
\end{array} \right)
\]

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 \text{terminate}, resulting in several MemoryMerge\text{TYP}_TYP actions, one for each type \text{TYP}_n.

\[
\text{MemoryMerge}\text{TYP}_1(b_{\text{TYP}_1}, ns) \equiv \\
\left( \begin{array}{c}
\varnothing (\text{terminate}) \parallel v : \text{dom} b_{\text{TYP}_1} \bullet \text{MemoryMerge}\text{TYP}_1 \text{Var}(v, b_{\text{TYP}_1}, ns)
\end{array} \right)
\]

\[
\text{MemoryMerge}(b_{\text{TYP}_1}, \ldots, b_{\text{TYP}_n}, ns) \equiv \\
\left( \begin{array}{c}
\text{MemoryMerge}\text{TYP}_1(b_{\text{TYP}_1}, ns) \\
\ldots \\
\varnothing (\text{terminate})
\end{array} \right)
\]

\[
\text{MemoryMerge}\text{TYP}_n(b_{\text{TYP}_n}, ns)
\]
Then, a second level of distribution is made within the scope of $\text{MemoryMergeTYP}_n$, where for every single state variable $v$ within the bindings $b_{\text{TYP}1}$, an action $\text{MemoryMergeTYP}_n \text{Var}$ 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.

$$\text{MemoryMergeTYP}_n \text{Var}(v, b_{\text{TYP}n}, ns) \equiv$$

$$\begin{cases} lget.v.(b_{\text{TYP}1}(v)) \rightarrow \text{MemoryMergeTYP}_1 \text{Var}(v, b_{\text{TYP}1}, ns) \\ \square lset.v?nv : typeTYP_1(n) \rightarrow \text{MemoryMergeTYP}_1 \text{Var}(v, b_{\text{TYP}1} \oplus \{v \mapsto nv\}, ns) \\ \square lterminate \rightarrow \begin{cases} \text{if } v \in ns \rightarrow (mset.v.(b_{\text{TYP}1}(v)) \rightarrow \text{Skip}) \\ v \not\in ns \rightarrow \text{Skip} \end{cases} \end{cases}$$

Regarding the $\text{Merge}$ action, it was literally merged into the $\text{MemoryMergeTYP}_n \text{Var}$ 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 $\text{MemoryMergeTYP}_n \text{Var}$.

Until now, we presented the elements used in the translation strategy in order to reshape a $\text{Circus}$ specification in such a way it can then be rendered in $\text{CSP}_M$ using the $\Upsilon$ functions. In the next section, we present details on the $\text{CSP}_M$ environment, as well as how we implemented the $\Upsilon$ functions.
Chapter 5

Translating Circus to $CSP_M$ with $\Upsilon$

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 ($\Upsilon$) 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 and 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 follow-
ing [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 specifica-
tion 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 $\{RAN.0, RAN.1, RAN.2, RAN.4, RAN.5\}$. 49
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`.

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.

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

Next, we define a series 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`.

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 $Memory(b_{RAN}, b_{ALA})$ 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.

\[
Memory_RAN Var(n, b_{RAN}) = \\
\text{2} \quad \text{mget.n} \cdot \text{apply}(b_{RAN}, n) \rightarrow Memory_RAN Var(n, b_{RAN}) \\
\text{3} \quad \text{[] mset.n?nv:RAN.eRAN(n)} \rightarrow Memory_RAN Var(n, over(b_{RAN}, n, nv)) \\
\text{4} \quad \text{[] terminate} \rightarrow \text{SKIP} \\
\text{Memory_RAN(b_{RAN})} = \\
\text{6} \quad \text{( [ | { terminate } ] | n : dom(b_{RAN}) } \\
\text{7} \quad \text{| Memory_RAN Var(n, address(n, b_{RAN})) }) \\
\text{Memory(b_{RAN}, b_{ALA}) = } \\
\text{Memory_ALA(b_{ALA}) [ | { terminate } | ] Memory_RAN(b_{RAN})}
\]

Likewise, the definitions for $MemoryMerge$ in $CSP_M$ are presented below. We make use of several auxiliary $CSP_M$ 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 omit the definitions for $U_{ALA}$ since they are similar to the ones for $U_{RAN}$.

\[
\begin{align*}
\text{MemoryMerge_RAN Var(n, b_{RAN}, ns)} = \\
\text{1} \quad \text{lget.n} \cdot \text{apply}(b_{RAN}, n) \rightarrow MemoryMerge_RAN Var(n, b_{RAN}, ns) \\
\text{2} \quad \text{[] lset.n?ty:RAN.eRAN(n)} \rightarrow MemoryMerge_RAN Var(n, over(b_{RAN}, n, nv), ns) \\
\text{3} \quad \text{[] lterminate} \rightarrow \text{SKIP} \\
\text{MemoryMerge_RAN(b_{RAN}, ns)} = \\
\text{5} \quad \text{( [ | { lterminate } ] | n : dom(b_{RAN}) } \\
\text{6} \quad \text{| MemoryMerge_RAN Var(n, address(n, b_{RAN})), ns) } \\
\text{7} \quad \text{| mset.n} \cdot \text{apply}(bd, n) \rightarrow \text{SKIP } \\
\text{MemoryMerge(b_{RAN}, b_{ALA}, ns) = } \\
\text{9} \quad \text{( MemoryMerge_ALA(b_{ALA}, ns) [ | { lterminate } | ] MemoryMerge_RAN(b_{RAN}, ns) )}
\end{align*}
\]

### 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 $OFF$.

\[
\begin{align*}
\text{b_{RAN1}} &= \{(sv_{sec}, RAN.0), (sv_{min}, RAN.0)\} \\
\text{b_{ALA1}} &= \{(sv_{buzz}, ALA.ON)\}
\end{align*}
\]

\(^1\)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-independently [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 \( \Upsilon_P \), used in order to render the structure of Circus processes in \( CSP_M \).

### 5.3 Mapping Circus Processes - \( \Upsilon_P \)

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

\[
\text{Proc}(b_{\text{RAN}}, b_{\text{ALA}}) = \\
\text{let} \quad \text{MemoryRANVar}(n, b_{\text{RAN}}) = \ldots \\
\quad \text{MemoryRAN}(b_{\text{RAN}}) = \ldots \\
\quad \text{MemoryALAVar}(n, b_{\text{ALA}}) = \ldots \\
\quad \text{MemoryALA}(b_{\text{ALA}}) = \ldots \\
\quad \text{MemoryALA}(b_{\text{ALA}}) = \left[ \left\{ \{ \text{terminate} \} \right\} \right] \text{MemoryRAN}(b_{\text{RAN}}) \\
\quad \ldots \\
\quad \text{MemoryMerge}(b_{\text{RAN}}, b_{\text{ALA}}) = \\
\quad \left[ \left\{ \text{terminate} \right\} \right] \text{MemoryMergeRAN}(b_{\text{RAN}}) \\
\text{within} \quad (\left( ( \text{MA} ; \text{terminate} -> \text{SKIP} ) \left\{ \text{MEMI} \right\} \text{Memory}(b_{\text{RAN}}, b_{\text{ALA}}) \right) \setminus \text{MEMI} )
\]

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 \( \Upsilon_A \) functions which will be presented in the next section.

### 5.4 Mapping Circus Actions - \( \Upsilon_A \)

The translation from Circus\(_{\text{CSP}} \) 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 \( \Omega \) functions transformation. Such a translation is made using the \( \Upsilon \) functions, which output the \( CSP_M \) text directly into a file.

\[
\Upsilon_A(c.v \rightarrow A) \equiv c.v \rightarrow \Upsilon_A(A)
\]

During our research we identified some inconsistency in the definition of the \( \Upsilon \) 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 \( \text{mget} \), a state variable \( v \) of a process \text{RingBuffer} should obtain the value \( v\text{RingCell}_v \) and should state that its type should match with the type of \text{RingBuffer}_v. As an example, a channel communication like \( \text{mget}.\text{RingCell}_v:v\text{RingCell}_v : \delta(\text{RingCell}_v) \) would not be captured by the two cases below.

\[
\Upsilon_A(c?x : P \rightarrow A) \equiv c?x : \{ x \mid x \leftarrow \delta(c), \, \Upsilon_A(P) \rightarrow \Upsilon_A(A) \\
\Upsilon_A(c?x \rightarrow A) \equiv c?x \rightarrow \Upsilon_A(A)
\]

52
Thus, there are not enough cases for the \( \Upsilon \) 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 \( \Upsilon \) function that covers the \textit{mget} and \textit{mset} communication, respectively, is provided below. We translate the \( \delta \) 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.

\[
\Upsilon_A(\text{mget}.x?\text{v}_x: \delta(x) \rightarrow A) \equiv \text{mget}.x?\text{v}_x:(\text{typeTYP}(x)) \rightarrow \Upsilon_A(A)
\]

\[
\Upsilon_A(\text{mset}.x.v_x \rightarrow A) \equiv \text{mset}.x.(\text{valueTYP}(v_x)) \rightarrow \Upsilon_A(A)
\]

From our example, the \textit{WakeUp} process, the \textit{mset} communication is then translated to the following extract, where the auxiliary functions are renamed suffixed by \( RAN \), the tag of \textit{sv._sec}.

\[
\ldots \rightarrow \text{mset}.\text{sv._sec}.(\text{RAN}.\text{valueRAN}(v_{sv._sec})) \rightarrow \text{SKIP}
\]

Similarly to process and action refinement, both the \( \Omega \) and \( \Upsilon \) functions are applied not only at the action level but also at the \textit{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 \textit{WakeUp} process

Moving to the main content of the specification, we use the example alarm clock, \textit{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 \textit{Circus} main action of the \textit{WakeUp} process.

\[
\text{WakeUp}(b_{RAN},b_{ALA}) =
\]

\[
\begin{align*}
\text{let } \text{MemoryRANVar}(n,b_{RAN}) & = \\
& \text{mget}.n.apply(b_{RAN},n) \rightarrow \text{MemoryRANVar}(n,b_{RAN}) \\
& \text{[{}]} \text{mset}.n?nv:\text{typeRAN}(n) \rightarrow \text{MemoryRANVar}(n,\text{over}(b_{RAN},n,nv)) \\
& \text{[{}]} \text{terminate} \rightarrow \text{SKIP} \\
& \text{MemoryRAN}(b_{RAN}) = \\
& \text{[{}]} \text{terminate} \land [{} n \land \text{dom}(b_{RAN}) \land \text{MemoryRANVar}(n,b_{RAN}) \\
& \text{Memory}(b_{RAN},b_{ALA}) = \text{MemoryALA}(b_{ALA}) \land [{} \land \text{terminate} \land [{}] \text{MemoryRAN}(b_{RAN}) \\
& \ldots \\
& \text{within } X \\
& \text{mset}.\text{sv._sec}.(\text{RAN}.0) \rightarrow \text{mset}.\text{sv._min}.(\text{RAN}.0) \rightarrow \text{mset}.\text{sv._buzz}.(\text{ALA.OFF}) \rightarrow \\
& (\text{let } X = \\
& \text{mget}.\text{sv._buzz}?v_{sv._buzz}:(\text{typeALA}(\text{sv._buzz})) \rightarrow \\
& \text{mget}.\text{sv._min}?v_{sv._min}:(\text{typeRAN}(\text{sv._min})) \rightarrow \\
& \text{mget}.\text{sv._sec}?v_{sv._sec}:(\text{typeRAN}(\text{sv._sec})) \rightarrow \\
& ( ( ( \\
& \text{tick} \rightarrow \\
& \text{mset}.\text{sv._sec}.(\text{RAN}.(\text{valueRAN}(v_{sv._sec}) + 1) \% 3) \rightarrow \\
& \text{mset}.\text{sv._min}.(\text{RAN}.(\text{valueRAN}(v_{sv._min}) + 1) \% 3) \rightarrow \\
& \text{mset}.\text{sv._sec}.(\text{RAN}.(\text{valueRAN}(v_{sv._sec}) + 1) \% 3) \rightarrow \\
& \text{mset}.\text{sv._min}.(\text{RAN}.(\text{valueRAN}(v_{sv._min}) == 1) \rightarrow \\
& \text{radioOn} \rightarrow \text{mset}.\text{sv._buzz}.(\text{ALA.ON}) \rightarrow \text{SKIP}) \\
& [{} ( (\text{valueRAN}(v_{sv._sec}) == 0) \land \\
& \text{mset}.\text{sv._min}.(\text{RAN}.(\text{valueRAN}(v_{sv._min}) + 1) \% 3) \rightarrow \\
& \text{mset}.\text{sv._sec}.(\text{RAN}.(\text{valueRAN}(v_{sv._sec}) == 1) \rightarrow \\
& \text{radioOff} \rightarrow \text{mset}.\text{sv._buzz}.(\text{ALA.OFF}) \rightarrow \text{SKIP)) \\
& [{} \text{time} \rightarrow \text{out}.(\text{valueRAN}(v_{sv._min}),\text{valueRAN}(v_{sv._sec}) \rightarrow \text{SKIP}) \\
& [{} \text{snooze} \rightarrow \text{mset}.\text{sv._buzz}.(\text{ALA.OFF}) \rightarrow \text{SKIP); X } \\
& \text{within } X) \\
& \text{terminate} \rightarrow \text{SKIP} \\
& [{} \text{MEMI} [{} \text{Memory}(b_{RAN},b_{ALA})] \land \text{MEMI} }
\end{align*}
\]
We can identify the original parts of the *Circus* specification in the *CSP* 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 an `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* 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, 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\textsubscript{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/

Figure 6.1: Circus2CSP initial screen

6.1 Commands for the translation to CSP\textsubscript{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)

\footnotetext[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 ($\Omega$); and (4) upsilon transformation ($\Upsilon$)

"reconv" repeats "conv"

"reset" cleans the buffer of loaded files

"quit" closes the application

---

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 \texttt{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 \texttt{\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 \texttt{assert} environment.

### 6.3 Integrating FDR4 with Circus2CSP

Model checking through FDR 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 \texttt{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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>procs</td>
<td>List all available processes</td>
</tr>
<tr>
<td>runfdr</td>
<td>Run FDR4 in command line mode</td>
</tr>
<tr>
<td>assert ref spec impl</td>
<td>Perform batch refinement for all processes available</td>
</tr>
<tr>
<td>assert ref spec impl model</td>
<td>Perform refinement using a given model</td>
</tr>
<tr>
<td>assert refall model</td>
<td>Perform all refinements</td>
</tr>
<tr>
<td>assert dl spec</td>
<td>Checks spec for dl</td>
</tr>
<tr>
<td>assert dl spec model</td>
<td>Checks spec for dl using a given model</td>
</tr>
<tr>
<td>assert dlall model</td>
<td>Perform 'dlall' using a given mode</td>
</tr>
<tr>
<td>assert div spec</td>
<td>Checks spec for div</td>
</tr>
<tr>
<td>assert div spec model</td>
<td>Checks spec for div using a given model</td>
</tr>
<tr>
<td>assert divsall</td>
<td>Perform batch div check for all processes available</td>
</tr>
<tr>
<td>assert divsall model</td>
<td>Perform 'divsall' using a given model</td>
</tr>
<tr>
<td>assert det spec</td>
<td>Checks if the spec is deterministic</td>
</tr>
<tr>
<td>assert det spec model</td>
<td>Checks if the spec is deterministic using a given model</td>
</tr>
<tr>
<td>assert jumbo</td>
<td>Perform all checks available (may take some time)</td>
</tr>
</tbody>
</table>

---

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 CSPM 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 CSPM 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.

NatValue == 0..5
DoorState ::= opened | closed
channel up, down, open, close

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.

\[
\text{process Lift } \triangleq \begin{align*}
\text{begin} \\
\text{state \ LiftState } & \triangleq [\text{floor} : \text{NatValue}; \ \text{doorState} : \text{DoorState}] \\
\text{InitLift } & \triangleq (\text{floor} := 0) \land (\text{doorState} := \text{closed}) \\
\text{Run } & \triangleq \begin{cases} \\
\text{((floor} < 5 \land \text{doorState} = \text{closed}) \land \text{up} \to (\text{floor} := \text{floor} + 1) \\
\text{((floor} > 0 \land \text{doorState} = \text{closed}) \land \text{down} \to (\text{floor} := \text{floor} - 1) \\
(\text{doorState} = \text{closed}) \land \text{open} \to (\text{doorState} := \text{opened}) \\
(\text{doorState} = \text{opened}) \land (\text{close} \to (\text{doorState} := \text{closed})) \\
\end{cases} \\
\bullet \text{InitLift} ; (\mu X \bullet (\text{Run} ; X)) \\
\end{align*}
\]

Then we use \textit{Circus2CSP} for translating the \textit{Lift} specification into \textit{CSP}$_M$, resulting in the following code.

1 \[
\text{Lift}(b_{\text{NAT}},b_{\text{DOO}}) = \\
\text{let} \\
\text{MemoryNATVar}(n,b_{\text{NAT}}) = \\
\{ (\text{mget.n.apply}(b_{\text{NAT}},n) \to \text{MemoryNATVar}(n,b_{\text{NAT}}) \\
\text{terminate} \to \text{SKIP}) \\
\text{MemoryDOOVar}(n,b_{\text{DOO}}) = \\
\text{MemoryNAT}(b_{\text{NAT}}) = (\{ \{ \text{terminate} \} \} \ n : \text{dom}(b_{\text{NAT}}) \land \text{MemoryNATVar}(n,b_{\text{NAT}}) ) \\
\text{MemoryDOO}(b_{\text{DOO}}) = \\
\text{Memory}(b_{\text{NAT}},b_{\text{DOO}}) = ( \text{MemoryDOO}(b_{\text{DOO}}) \land \text{MemoryNAT}(b_{\text{NAT}}) ) \\
\text{within} ( ( \\
\text{mset.sv_floor.(NAT.0) \to \text{mset.sv_doorState.(DOO.closed)} \to \\
\text{let} X = \text{mget.sv_doorState?v_sv_doorState:(typeDOO(sv_doorState)) \to} \\
\{ (\{ ( \\
(\text{valueNAT}(v_{\text{sv_floor}}) < 5) \land \text{valueDOO}(v_{\text{sv_doorState}}) == \text{closed}) \\
\text{up} \to \text{mset.sv_floor.(NAT.(valueNAT(v_{\text{sv_floor}}) + 1)) \to \text{SKIP} ) \\
(\{ (\text{valueNAT}(v_{\text{sv_floor}}) > 0) \land \text{valueDOO}(v_{\text{sv_doorState}}) == \text{closed}) \\
\text{down} \to \text{mset.sv_floor.(NAT.(valueNAT(v_{\text{sv_floor}}) - 1)) \to \text{SKIP} ) \\
(\{ \text{valueDOO}(v_{\text{sv_doorState}}) == \text{closed}) \\
\text{open} \to \text{mset.sv_doorState.(DOO.opened) \to \text{SKIP} ) \\
(\{ \text{valueDOO}(v_{\text{sv_doorState}}) == \text{opened}) \\
\text{close} \to \text{mset.sv_doorState.(DOO.closed) \to \text{SKIP} ); \\
X ) \land \text{within X}; \\
\text{terminate} \to \text{SKIP} ) \\
\text{EMEM} \} \land \text{Memory}(b_{\text{NAT}},b_{\text{DOO}}) \land \text{EMEM} ) \\
\text{then} \\
\text{from the translation from Beg’s tool,} \\
\text{INITLIFT}(\text{doorState},\text{floor}) = \text{LIFT}(\text{closed},0) \\
\text{LIFT}(\text{doorState},\text{floor}) = \\
(\{ ( \{ \\
(\text{floor} < 5) \land \text{doorState} == \text{closed}) \land (\text{up} \to \text{LIFT}(\text{doorState},\text{floor} + 1) \\
(\{ (\text{floor} > 0) \land \text{doorState} == \text{closed}) \land (\text{down} \to \text{LIFT}(\text{doorState},\text{floor} - 1) \\
(\{ \text{doorState} == \text{closed}) \land (\text{open} \to \text{LIFT}(\text{opened},\text{floor}) \\
(\{ \text{doorState} == \text{opened}) \land (\text{close} \to \text{LIFT}(\text{closed},\text{floor}) \\
\text{We compare both models by defining \textit{ArshadLift} and \textit{CTOCLift} processes, with the same binding values, and check for the refinement between them, which turns out to be equivalent to each other.} \\
1 \[
\text{ArshadLift} = \text{INITLIFT}(\text{valueDOO}(\text{apply}(b_{\text{DOO1}},\text{sv_doorState})), \\
\text{valueNAT}(\text{apply}(b_{\text{NAT1}},\text{sv_floor}))) \\
\text{CTOCLift} = \text{Lift}(b_{\text{NAT1}},b_{\text{DOO1}}) \\
\text{assert} \text{ArshadLift} \ [\text{FD} \text{= CTOCLift} \ -- \text{Passed} \\
\text{assert} \text{CTOCLift} \ [\text{FD} \text{= ArshadLift} \ -- \text{Passed} \\
\text{Example 2} – \text{Simple Sequential Chain of Calls in Initialiser}
In this example, assignments are combined with channel communication, in a sequential composition. 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.

\[
\text{channel } a, b, c \\
\text{process } \text{ProcEx}_1 \equiv \\
\begin{align*}
\text{begin} & \\
\text{state } \text{Ex}_1\text{State} & \equiv [x, y : \text{NatValue}] \\
\text{InitProcEx}_1 & \equiv (x := 1); \ (y := 1) \\
\text{Action}_A & \equiv (x := 0); \ (a \rightarrow \text{Skip}) \\
\text{Action}_B & \equiv (y := x + 1 \mod 5); \ (b \rightarrow \text{Skip}) \\
\text{Action}_C & \equiv c \rightarrow \text{Skip} \\
& \cdot \text{Action}_C; \ \text{Action}_A; \ \text{Action}_B
\end{align*}
\]

Then we use \textit{Circus2CSP} for translating the \textit{ProcEx}_1 specification into \textit{CSP}_M, resulting in the following code.

\begin{verbatim}
1 ProcEx1(b_NAT) = let
3   MemoryNATVar(n,b_NAT) = ...
5   Memory(b_NAT) = MemoryNAT(b_NAT)
7   within ( ( (
9   mset.sv_x.(NAT.0) -> a --> -- ActionA
11   mset.sv_y.(NAT.((valueNAT(v_sv_x) + 1)%5)) -> b --> SKIP; -- ActionB
13   terminate --> SKIP )
15   | MEMI | Memory(b_NAT))
17 )
\end{verbatim}

The translation provided by Beg is:

\begin{verbatim}
ACTION_C(x,y) = c --> ACTION_A(x,y)
ACTION_A(x,y) = a --> ACTION_B(0,y)
ACTION_B(x,y) = b --> SKIP
\end{verbatim}

Then we define the two \textit{CSP}_M processes \textit{ArshadProEx}_1 and \textit{CTOCProcEx}_1, and check the refinement between them, proving its equivalence.

\begin{verbatim}
1 ArshadProEx1 = ACTION_C(valueNAT(apply(b_NAT1,sv_x)),
2   valueNAT(apply(b_NAT1,sv_y)))
3 CTOCProcEx1 = ProcEx1(b_NAT1)
4 assert ArshadProEx1 [FD= CTOCProcEx1 -- Passed
5 assert CTOCProcEx1 [FD= ArshadProEx1 -- Passed
\end{verbatim}

\section*{Example 3 – Main Action having Internal Choice}

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

\[
\text{process } \text{ProcEx}_2 \equiv \\
\begin{align*}
\text{begin} & \\
\text{state } \text{Ex}_2\text{State} & \equiv [x, y : \text{NatValue}] \\
\text{InitProcEx}_2 & \equiv (x := 1); \ (y := 1) \\
\text{Action}_A & \equiv (x := 0); \ (a \rightarrow \text{Skip}) \\
\text{Action}_B & \equiv (y := x + 1 \mod 5); \ (b \rightarrow \text{Skip}) \\
\text{Action}_C & \equiv c \rightarrow \text{Skip} \\
& \cdot \text{Action}_A; \ (\text{Action}_B \sqcap \text{Action}_C)
\end{align*}
\]

61
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 $\text{InitProcEx}2$, its behaviour is not translated into $\text{CSP}_M$. Therefore, the $\text{CSP}_M$ code of $\text{ProcEx}2$ is:

```plaintext
1 ProcEx2(b_NAT) =
2   let
3       MemoryNATVar(n, b_NAT) = ...
4       MemoryNAT(b_NAT) = ...
5   Memory(b_NAT) = MemoryNAT(b_NAT)
6 within ( ( ( mset.sv_x.(NAT_0) ->
7           a ->
8           mget.sv_x?sv_x:(typeNAT(sv_x) ->
9               mset.sv_y.(NAT.{valueNAT(v_sv_x) + 1|sv_x}) ->
10              b -> SKIP
11           |~| mset.sv_x.(NAT_0) -> a -> c -> SKIP );
12                       terminate -> SKIP )
13 ( [ | MEMI | ] Memory(b_NAT))\MEMI )
```

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

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

Which is indeed the same behavior as of Beg’s translated version $\text{ACTION_FINAL}$:

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

And therefore, both processes are equivalent.

```plaintext
1 ArshadProcEx2 = ACTION_FINAL(valueNAT(apply(b_NAT1,sv_x)),valueNAT(apply(b_NAT1,sv_y)))
2 CTOCProcEx2 = ProcEx2(b_NAT1)
3 assert ArshadProcEx2 [FD= CTOCProcEx2 -- Passed
4 assert CTOCProcEx2 [FD= ArshadProcEx2 -- Passed
```

Example 4 – Multiple assignments in an Action

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

```plaintext
process ProcEx4 $\triangleq$
begin
  state Ex4State $\triangleq \{x, y : \text{NatValue}\}$
  InitProcEx4 $\triangleq (x := 1); (y := 1)$
  ActionA $\triangleq (x := 0); (a \rightarrow \text{Skip})$
  ActionB $\triangleq (y := x + 1 \pmod{5}); (b \rightarrow \text{Skip})$
    \begin{align*}
      (x & := y + 1 \pmod{5}) \\
      (y & := y + x \pmod{5})
    \end{align*}
  ActionC $\triangleq$ \begin{align*}
    (x & := 4); \\
    (y & := 5); \\
    (c \rightarrow \text{Skip})
  \end{align*}
  $\bullet$ $(\text{ActionA}; (\text{ActionB} ; \text{ActionC}))$
end
```
We can see that some assignments do not mention any state variable, whereas others like \( x := y + 1 \) do require the values of such variables. Therefore, sometimes a \textit{mget} event occurs prior to the \textit{mset} event, for retrieving the current value of the variables mentioned in the assignment.

```plaintext
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)) ->
               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 }
            [ "MEME"
            Memory(b_NAT))"MEME]
```

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 \textit{ActionA}, we first have an assignment, \( x := 0 \), then an event \( a \) followed by \textit{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:

```plaintext
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 \textit{ActionB} behaves as described in \textit{ActionA}. However, for \textit{ActionC}, all assignments, and therefore, state transitions, occurs prior to the event \( c \), which is the final one, \textit{ACTION_C_4}, as described above.

Our approach using \textit{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.

```plaintext
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
```

\textbf{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.

\[
\begin{align*}
\text{process } \text{ProcEx5} & \triangleq \\
\text{begin} & \\
\text{state } \text{Ex5State} & \triangleq [x, y : \text{NatValue}] \\
\text{InitProcEx5} & \triangleq (x := 1); (y := 1) \\
\text{ActionA} & \triangleq (x := 0); (a \rightarrow \text{Skip}) \\
\text{ActionB} & \triangleq (y := x + 1 \mod 5); (b \rightarrow \text{Skip}) \\
\text{ActionC} & \triangleq c \rightarrow \text{Skip} \\
\bullet \text{ActionA} & ; (\text{ActionB } \land \text{ActionC}) \\
\end{align*}
\]

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

\[
\text{ProcEx5}(... ; (((y := x + 1 \mod 5); b \rightarrow \text{Skip}) \sqcap (c \rightarrow \text{Skip})))
\]

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 \textit{Circus} refinement laws combined with the omega transformations. As we know that assignments are not available in \textit{CSP}_M, we translate them into a sequence of \textit{mget} and \textit{mset} events, which are hidden to the external environment. Using the refinement laws we were able to shift the \textit{mget} and \textit{mset} events to occur after the event choice. Therefore, based on the Proof G.2.2, the above external choice becomes:

\[
\text{ProcEx5}(... ; (((b \rightarrow \text{mset.sv_y.((v_{sv_x} + 1) \mod 5} \rightarrow \text{Skip}) \sqcap (c \rightarrow \text{Skip})))
\]

Thus, as expected, the \textit{CSP}_M code of the \textit{ProcEx5} translates the \textit{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 \textit{b} and \textit{c} and depending on them, the value of \textit{y} may be updated.

\[
\begin{align*}
\text{ProcEx5(b_{NAT})} & = \\
& \text{let} \\
& \text{MemoryNATVar}(n, b_{\text{NAT}}) = ... \\
& \text{MemoryNAT}(b_{\text{NAT}}) = ... \\
& \text{Memory}(b_{\text{NAT}}) = \text{MemoryNAT}(b_{\text{NAT}}) \\
& \text{within} ( ( ( \\
& \text{mset.sv_x.((NAT.0) \rightarrow a \rightarrow -- ActionA} \\
& \text{mget.sv_x?v_{sv_x}.(\text{NAT.(v_{sv_x})}) \rightarrow} \\
& \text{mget.sv_y?v_{sv_y}.(\text{NAT.(v_{sv_y})}) \rightarrow} \\
& ( \\
& \text{b \rightarrow mset.sv_y.((v_{sv_y} + 1) \mod 5} \rightarrow \text{Skip} \rightarrow -- \text{ActionB} \\
& \text{c \rightarrow SKIP \rightarrow -- \text{ActionC} \\
& \text{terminate \rightarrow SKIP } \\
& ) [\text{MEMI}] \text{Memory}(b_{\text{NAT}}))\text{MEMI} ) \\
\end{align*}
\]

Finally, as expected, both translations of \textit{ProcEx5} are equivalent to each other.

\[
\begin{align*}
\text{ACTION_FINAL5(y,x)} & = a \rightarrow \text{BOX}(y,0) \\
\text{BOX}(y,x) & = b \rightarrow \text{SKIP} [] c \rightarrow \text{SKIP} \\
\text{ArshadProEx5} & = \text{ACTION_FINAL5}(\text{valueNAT(apply(b_{\text{NAT1}},sv_x)))}
\end{align*}
\]

\(^1\)The entire proof can be found in the Appendix G.2
Example 6 – Assignment Does Not Resolve Choice - Assignment 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 occurrence of the events, whichever was chose.

$$\text{process } ProcEx6 \triangleq$$
$$\begin{align*}
\text{state } Ex6State &\triangleq [x, y : \text{NatValue}] \\
InitProcEx6 &\triangleq (x := 1); (y := 1) \\
ActionA &\triangleq (x := 0); (a \rightarrow \text{Skip}) \\
ActionB &\triangleq (y := x + 1); (b \rightarrow \text{Skip}) \\
ActionC &\triangleq c \rightarrow \text{Skip} \\
\bullet (ActionA \Box ActionB); ActionC
\end{align*}$$

Therefore, using Circus2CSP, we obtain the following CSPM 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)) \rightarrow mget.sv_y?v_sv_y:(typeNAT(sv_y)) \rightarrow 
  ( 
    a \rightarrow mset.sv_x.(NAT.0) \rightarrow \text{SKIP} -- ActionA 
    b \rightarrow mset.sv_y.(NAT.(valueNAT(v_sv_x) + 1)%5)) \rightarrow \text{SKIP} -- ActionB 
  ;
  c \rightarrow \text{SKIP} -- ActionC 
  ) ;
.terminate \rightarrow \text{SKIP} )
[| MEMI | Memory(b_NAT)\MEMI ]
```

The resulting CSPM 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,

$$\begin{align*}
(ActionA \Box ActionB); ActionC \\
\subseteq (ActionA; ActionC) \Box (ActionB; ActionC)
\end{align*}$$
$$\begin{align*}
\subseteq \bigg( & (x := 0; a \rightarrow \text{Skip}); c \rightarrow \text{Skip} \bigg) \\
& \bigg( (y := x + 1; b \rightarrow \text{Skip}); c \rightarrow \text{Skip} \bigg)
\end{align*}$$

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

\[
\text{HDGenComp} \equiv [\text{hdActivity} : \mathbb{P} \text{ACTIVITY} \land \text{infSalVol} : \mathbb{N} \land \ldots]
\]

We now define the Circus process HDMachine, which captures all the state variables in “state HD-State”, 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].

\[
\text{process HDMachine} \equiv \begin{align*}
\text{begin} \\
\text{state HDState} &= \begin{cases}
\text{HDGenComp} \land \text{RinsingParameters} \\
\land \text{DFParameters} \land \text{UFParameters} \\
\land \text{PressureParameters} \land \text{HeparinParameters}
\end{cases}
\end{align*}
\]

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.

\[
\text{MainTherapy} \equiv \text{TherapyPreparation} ; \text{TherapyInitiation} ; \text{TherapyEnding}
\]

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

![Figure 7.1: Execution flow of the Haemodialysis machine](attachment:image.png)
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].

\[
\text{TherapyPreparation} \triangleq \\
\text{preparationPhase } \rightarrow \text{AutomatedSelfTest; \hfill ConnectingTheConcentrate; \hfill SetRinsingParameters; \hfill InsertingRinsingTestingTubSystem; \hfill PrepHeparinPump; \hfill SetTreatParameters; \hfill RinsingDialyzer}
\]

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:

\[
\text{During the application of arterial bolus, the system monitors the volume of saline infusion and if the volume exceeds 400ml, the system should stop the blood flow and raise an alarm signal.}
\]

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.

\[
R1 \triangleq \left\{ \begin{array}{l}
\text{if} \\
\left( \text{hdActivity} = \{\text{appArtBolus}\} \land \text{infSalVol} > 400 \right) \\
\rightarrow (\text{StopBloodFlow} \parallel \text{RaiseAlarm}) \\
\lor \left( \neg \left( \text{hdActivity} = \{\text{appArtBolus}\} \land \text{infSalVol} > 400 \right) \\
\rightarrow (\text{Wait(CheckInterval)} \rightarrow R1)
\end{array} \right.
\fi
\]

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.

\[
\text{HDMain} \triangleq \text{HDGentCompInit} ; \ (\text{MainTherapy} \parallel R1)
\]

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 CSPM model of the Haemodialysis machine as presented in
ABZ’16, using byHand. First, we translate the Z types into equivalent constructs in CSPM. 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 CSPM. 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.

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

Then, Z schemas are translated by defining name types in CSPM, 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
\( \mathbb{P} \text{ACTIVITY} \), translated into CSPM 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.

```plaintext
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 CSPM is the fact that manipulating state variables is not
possible in CSPM 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 \( \text{hdActivity} \), 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 \( \text{hdActivity} := \text{val} \) 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.

69
The overall shape of the \textit{HDMachine} process translated using \textit{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 \textit{Circus} state is defined as the input of that process in CSP\textsubscript{M} and the access to those values is made through CSP\textsubscript{M} channels, allowing the main action of the process to access and update the values of the state components.

For instance, the \textit{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 \textit{HDGenComp}: each variable is read using the channels prefixed with \textit{get}, and is updated through channels prefixed with \textit{set}. For instance, \textit{getHdActivity} provides the current value of \textit{hdActivity}, and \textit{setHdActivity} receives the its value and recurses updating it in HDGC.

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

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

MainTherapy = TherapyPreparation ; TherapyInitiation ; TherapyEnding

The structure of the CSP\textsubscript{M} process is defined as the execution of HDGenCompInit, initialising the HDGenComp variables, followed by the MainTherapy in parallel with R1. 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.

\begin{verbatim}
within ((HDGenCompInit ; |R1 ||| MainTherapy))
   [[HDComm]] HDGenCompSt(HDGC) \ HDComm
\end{verbatim}

This concludes the translation of the haemodialysis machine using \textit{byHand}. The resulting CSP\textsubscript{M} file has twice the number of lines compared to its \textit{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 CSPM process. In the next subsection, we present the translated model using Circus2CSP.

**Automatic Translation using Circus2CSP**

We present now a short description of the CSPM 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)
    [] terminate -> SKIP
    MemoryNAT(b_NAT) =
      [{terminate | n : dom(b_NAT) @ MemoryNATVar(n,b_NAT)}
      ... Memory(b_NAT,...) = (MemoryNAT(b_NAT) [{terminate |}] Memory...)
    within | { }
    preparationPhase -> autSelfTest -> -- TherapyPreparation action
    mset.sv_signalLamp.(BOO.TURE) ->
      connectingConcentrate'x -> mset.sv_bicarbonateAcetate.(BOO.x) ->
      { { { { atrialTubing -> SKIP ||| ventricularTubing -> SKIP };
          salineBagLevel?ifs ->
          mset.sv_infSalineVol.(NAT.ifs) -> SKIP };
          insertHeparinSyringe ->
          heparinLinesIsVented -> SKIP };
        fillArterialDrip -> stopBP -> SKIP };
        therapyInitiation -> connectingToPatient ->
        mset.sv_signalLamp.(BOO.FALSE) ->
        connPatientArterially ->
        setbloodFlowInEBC?bf -> mset.sv_bloodFlowInEBC.(NAT.bf) ->
        connPatientVenously ->
        mset.sv_signalLamp.(BOO.TURE) ->
        ... mset.sv_signalLamp.(BOO.FALSE) ->
        therapyEnding -> -- TherapyEnding action
        mset.sv_signalLamp.(BOO.FALSE) -> SKIP |
        { { let muR1 = -- Begin of R-1
            mset.sv_hdActivity?v_sv_hdActivity:(typePRO(sv_hdActivity)) ->
            mset.sv_infSalineVol?v_sv_infSalineVol:(typeNAT(sv_infSalineVol)) ->
            { (valuePRO(v_sv_hdActivity) == [TRUE])
              and (valueNAT(v_sv_infSalineVol) > 0)) &
            { stopBloodFlow -> SKIP };
          | mset.sv_alarm.(BOO.TRUE) -> produceAlarmSound -> SKIP };
          | not { (valuePRO(v_sv_hdActivity) == [TRUE])
            and (valueNAT(v_sv_infSalineVol) > 0)) &
          | tick -> muR1 }
        within | { MEMI |} End of R-1
      }; terminate -> SKIP |
      { MEMI |} Memory(b_PRO,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 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 file doubled (2264 lines) compared to the \texttt{\LaTeX} Circus specification (1024 lines). Moreover, 124 new auxiliary functions (\texttt{get} and \texttt{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 leads to a type error in FDR as it does not support polymorphic function definitions, which is the case of the \texttt{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 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 \texttt{mset} and \texttt{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 \texttt{mget} and \texttt{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:

\[ NatValue == 0 \ldots N \]
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

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

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 . . . 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_{241}). 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.

<table>
<thead>
<tr>
<th>Test</th>
<th>Model</th>
<th>Result</th>
<th>States Visited</th>
<th>Transitions</th>
<th>Plys</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>deadlock free</td>
<td>RB_{241}</td>
<td>Passed</td>
<td>8,297,025</td>
<td>16,805,249</td>
<td>44</td>
<td>26.657s</td>
</tr>
<tr>
<td></td>
<td>RB_{CTOC}</td>
<td>Passed</td>
<td>1,628</td>
<td>3,109</td>
<td>38</td>
<td>0.145s</td>
</tr>
<tr>
<td></td>
<td>RB_{KW}</td>
<td>Failed</td>
<td>248</td>
<td>713</td>
<td>8</td>
<td>0.128s</td>
</tr>
<tr>
<td>deterministic</td>
<td>RB_{241}</td>
<td>Passed</td>
<td>9,869,889</td>
<td>19,852,673</td>
<td>69</td>
<td>54.863s</td>
</tr>
<tr>
<td></td>
<td>RB_{CTOC}</td>
<td>Passed</td>
<td>2,012</td>
<td>3,853</td>
<td>63</td>
<td>0.159s</td>
</tr>
<tr>
<td></td>
<td>RB_{KW}</td>
<td>Failed</td>
<td>27</td>
<td>84</td>
<td>3</td>
<td>0.156s</td>
</tr>
</tbody>
</table>

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 determinism. 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_F D 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_F D RB_{CTOC} and RB_{CTOC} \sqsubseteq_F D 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

<table>
<thead>
<tr>
<th>Assertion</th>
<th>States Visited</th>
<th>Transitions Visited</th>
<th>Plys</th>
<th>Exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 RB_{byH} \sqsubseteq_F D RB_{CTOC}</td>
<td>✓</td>
<td>1,628</td>
<td>3,109</td>
<td>38</td>
</tr>
<tr>
<td>2 RB_{KW} \sqsubseteq_F D RB_{CTOC}</td>
<td>✓</td>
<td>1,733</td>
<td>3,365</td>
<td>43</td>
</tr>
<tr>
<td>3 RB_{CTOC} \sqsubseteq_F D RB_{byH}</td>
<td>✓</td>
<td>8,297,025</td>
<td>16,805,249</td>
<td>44</td>
</tr>
<tr>
<td>4 RB_{KW} \sqsubseteq_F D RB_{byH}</td>
<td>✓</td>
<td>8,809,345</td>
<td>18,087,809</td>
<td>49</td>
</tr>
<tr>
<td>5 RB_{byH} \sqsubseteq_F D RB_{KW}</td>
<td>×</td>
<td>27</td>
<td>98</td>
<td>3</td>
</tr>
<tr>
<td>6 RB_{CTOC} \sqsubseteq_F D RB_{KW}</td>
<td>×</td>
<td>27</td>
<td>91</td>
<td>3</td>
</tr>
</tbody>
</table>

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_F D 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 $\text{CSP} \mid \mu \text{B}$. We observed that the use of $\text{subtype}$, 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 techniques, such as $\text{sbisim}$, which determines the maximal strong bisimulation [25], and $\text{wbisim}$, 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>
<thead>
<tr>
<th>Values Range</th>
<th>States Visited</th>
<th>Exec Time (seconds)</th>
<th>States Visited</th>
<th>Exec Time (seconds)</th>
<th>States Visited</th>
<th>Exec Time (seconds)</th>
<th>States Visited</th>
<th>Exec Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..10</td>
<td>77</td>
<td>0.499</td>
<td>21,169</td>
<td>0.458</td>
<td>302</td>
<td>0.479</td>
<td>87</td>
<td>0.56</td>
</tr>
<tr>
<td>0..20</td>
<td>77</td>
<td>0.986</td>
<td>74,949</td>
<td>0.819</td>
<td>302</td>
<td>0.925</td>
<td>87</td>
<td>1.106</td>
</tr>
<tr>
<td>0..40</td>
<td>77</td>
<td>2.901</td>
<td>280,909</td>
<td>2.263</td>
<td>302</td>
<td>2.582</td>
<td>87</td>
<td>3.647</td>
</tr>
<tr>
<td>0..80</td>
<td>77</td>
<td>11.093</td>
<td>1,086,429</td>
<td>8.043</td>
<td>302</td>
<td>10.013</td>
<td>87</td>
<td>13.343</td>
</tr>
<tr>
<td>0..120</td>
<td>77</td>
<td>25.096</td>
<td>2,416,749</td>
<td>18.805</td>
<td>302</td>
<td>21.793</td>
<td>87</td>
<td>35.839</td>
</tr>
<tr>
<td>0..160</td>
<td>77</td>
<td>47.635</td>
<td>4,271,869</td>
<td>35.148</td>
<td>302</td>
<td>42.267</td>
<td>87</td>
<td>70.011</td>
</tr>
<tr>
<td>0..240</td>
<td>77</td>
<td>114.845</td>
<td>9,556,509</td>
<td>84.803</td>
<td>302</td>
<td>100.112</td>
<td>87</td>
<td>175.846</td>
</tr>
<tr>
<td>0..360</td>
<td>77</td>
<td>327.815</td>
<td>21,419,469</td>
<td>235.236</td>
<td>302</td>
<td>269.414</td>
<td>killed</td>
<td>286.079</td>
</tr>
<tr>
<td>0..480</td>
<td>77</td>
<td>668.437</td>
<td>38,005,629</td>
<td>467.602</td>
<td>302</td>
<td>523.825</td>
<td>killed</td>
<td>525.889</td>
</tr>
</tbody>
</table>

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 $\text{NatValue}$ 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 $\text{wbisim}$ for the cases where the range of $\text{NatValue}$ 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 \( \text{second} : TIME \), if the variable \( \text{second} \) is used once in the entire specification, lets us consider the expression \( \text{second} > 15 \) 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 \( \text{second} : \{15,16\} \). 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 MemoryD241) and the partitioned memory process used in CTOC (MemoryCTOC), as presented below.

\[
\text{MemoryD241} \triangleq \text{vres } b : \text{BINDING } \\
\left( (\square n : \text{dom } b \cdot \text{mget}.n!b(n) \rightarrow \text{MemoryD241}(b)) \right) \\
\left( (\square n : \text{dom } b \cdot \text{mset}.n?nv : (nv \in \delta(n)) \rightarrow \text{MemoryD241}(b \oplus \{n \mapsto nv\})) \right) \\
\square \text{terminate} \rightarrow \text{Skip}
\]

\[
\text{MemoryCTOC} \triangleq \text{var } b : \text{BINDING } \\
\left( [\square \text{terminate}] [ ] n : \text{dom } b \cdot \text{MemoryVar}(n, n \triangleleft b) \right)
\]

\[
\text{MemoryVar} \triangleq \text{var } n : \text{NAME}; b : \text{BINDING } \\
\left( \text{mget}.n!b(n) \rightarrow \text{MemoryVar}(n, b) \right) \\
\left( \square \text{mset}.n?nv : (nv \in \delta(n)) \rightarrow \text{MemoryVar}(n, (b \oplus \{n \mapsto nv\})) \right) \\
\square \text{terminate} \rightarrow \text{Skip}
\]

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

\[
\text{MemoryD241} \sqsubseteq \text{MemoryCTOC}
\]

and

\[
\text{MemoryCTOC} \sqsubseteq \text{MemoryD241}
\]

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.
\[\text{Memory}_{\text{CTOC}}(b)\]

\[[\text{Memory}_{\text{CTOC}} \text{ definition}]

\[\square \text{ terminate} \] \(n : \text{dom } b \bullet \text{Memory}_{\text{Var}}(n, n < b)\)

\[[\text{Iterated parallelism definition}]

\[\text{Memory}_{\text{Var}}(n_1, n_1 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_2, n_2 < b)\]

\[[\text{Memory}_{\text{Var}} \text{ definition}]

\[\begin{align*}
\text{mget}.n_1!b(n_1) & \rightarrow \text{Memory}_{\text{Var}}(n_1, n_1 < b) \\
\text{mget}.n_2!b(n_2) & \rightarrow \text{Memory}_{\text{Var}}(n_2, n_2 < b)
\end{align*}\]

\[\square \text{ terminate} \rightarrow \text{Skip}\]

\[[82, L7 - p.51]\]

\[\begin{align*}
\text{mget}.n_1!b(n_1) & \rightarrow \left( \text{Memory}_{\text{Var}}(n_1, n_1 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_2, n_2 < b) \right) \\
\text{mget}.n_2!b(n_2) & \rightarrow \left( \text{Memory}_{\text{Var}}(n_2, n_2 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_1, n_1 < b) \right)
\end{align*}\]

\[\square \text{ terminate} \rightarrow \text{Skip} \rightarrow \text{Skip}\]

\[[\text{Parallelism Composition Unit - [129, Law C.90]}]

\[\begin{align*}
\text{mget}.n_1!b(n_1) & \rightarrow \left( \text{Memory}_{\text{Var}}(n_1, n_1 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_2, n_2 < b) \right) \\
\text{mget}.n_2!b(n_2) & \rightarrow \left( \text{Memory}_{\text{Var}}(n_2, n_2 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_1, n_1 < b) \right)
\end{align*}\]

\[\square \text{ terminate} \rightarrow \text{Skip} \rightarrow \text{Skip}\]

\[\begin{align*}
\text{mget}.n_1!b(n_1) & \rightarrow \left( \text{Memory}_{\text{Var}}(n_1, n_1 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_2, n_2 < b) \right) \\
\text{mget}.n_2!b(n_2) & \rightarrow \left( \text{Memory}_{\text{Var}}(n_2, n_2 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_1, n_1 < b) \right)
\end{align*}\]

\[\text{mset}.n_1?nv : (nv \in \delta(n_1)) \rightarrow \left( \text{Memory}_{\text{Var}}(n_1, n_1 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_2, n_2 < b) \right)\]

\[\text{mset}.n_2?nv : (nv \in \delta(n_2)) \rightarrow \left( \text{Memory}_{\text{Var}}(n_2, n_2 < b) \parallel \square \text{ terminate} \parallel \text{Memory}_{\text{Var}}(n_1, n_1 < b) \right)\]

\[\text{terminate} \rightarrow \text{Skip}\]

\[\begin{align*}
\text{mget}.n_1!b(n_1) & \rightarrow \text{Memory}_{\text{CTOC}}(b) \\
\text{mget}.n_2!b(n_2) & \rightarrow \text{Memory}_{\text{CTOC}}(b) \\
\text{mset}.n_1?nv : (nv \in \delta(n_1)) & \rightarrow \text{Memory}_{\text{CTOC}}(b \oplus \{n_1 \mapsto nv\}) \\
\text{mset}.n_2?nv : (nv \in \delta(n_2)) & \rightarrow \text{Memory}_{\text{CTOC}}(b \oplus \{n_2 \mapsto nv\})
\end{align*}\]

\[[\text{Iterated External Choice definition}]

\[\begin{align*}
\square n : \text{dom } b \bullet \text{mget}.n!b(n) & \rightarrow \text{Memory}_{\text{CTOC}}(b) \\
\square \square n : \text{dom } b \bullet \text{mset}.n?nv : (nv \in \delta(n)) & \rightarrow \text{Memory}_{\text{CTOC}}(b \oplus \{n \mapsto nv\})
\end{align*}\]

\[\text{terminate} \rightarrow \text{Skip}\]

78
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 CSPM 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 CSPM description of MemoryD241 and RHS is the equivalent model to MemoryCTOC.

We illustrate the CSPM 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
|| {terminate())|
--
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 S4 [FD= RHS
assert RHS [FD= S4
```
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, as mentioned in Section 3.1, we want to prove that the refinement steps implemented in Haskell, herein \( \Omega_H \), from staterich Circus processes (CircusSR) to stateless Circus processes (CircusSL) are correct w.r.t. those refinement steps implemented in Isabelle/UTP, represented here as \( \Omega_I \). We highlight that the notation used here for \( \Omega_H \) and \( \Omega_I \) represents not only the \( \Omega \) 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 therefore, 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 \texttt{\LaTeX} would still take time. Finally, we would either way have to export CSP\textsubscript{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\textsubscript{M}, we noticed that because assignments are transformed into a sequence of events (\textit{mget} and \textit{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.

\[
\begin{align*}
\Omega(\muS.((x := 0 ; c_1 \rightarrow \text{Skip}) \sqcap (x := 1 ; c_2 \rightarrow \text{Skip}))) \\
= \\
\muS.((c_1 \rightarrow x := 0) \sqcap (c_2 \rightarrow x := 1))
\end{align*}
\]

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 \textit{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 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 – π 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 – π 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 CSPM [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 CSPM 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, stating if the assertion has passed or not. We plan for the future to parse the counter-examples provided by FDR into our tool.

\section{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 TEX 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


[144] RTCA Special Committee 190 and EUROCAE Working Group 52. Errata DO-178B. Software considerations in airborne systems and equipment certification, 1999.


Appendix A

Circus AST

A.1 Circus and Z Paragraphs

\[
ZPara ::= \text{channel } CDecl \\
| \text{channelset } N == CSExp \\
| \text{ProcDecl}
\]

A.2 Circus Channel Declaration – \textit{CDecl}

\[
CDecl ::= n \\
| n : Exp \\
| n[e] : Exp
\]

A.3 Circus Process – \textit{ProcDecl}

\[
ProcDecl ::= \text{process } N \equiv \text{ProcDef} \\
| \text{process } N(N^+) \equiv \text{ProcDef} \\
| \text{process } N[N^+] \equiv \text{ProcDef}
\]

A.4 Circus Process – \textit{ProcessDef}

\[
ProcessDef ::= \text{Decl } \bullet \text{ProcDef} \\
| \text{Decl } \odot \text{ProcDef} \\
| \text{Proc}
\]
\section*{A.5 Circus Process \textemdash CProc}

\begin{aligned} 
CProc &::= \quad N \\
& \mid N[Exp^+] \\
& \mid N(Exp^+) \\
& \mid N\lceil Exp^+\rfloor \\
& \mid ; \quad \text{Decl} \bullet \text{CProc} \\
& \mid \Box \text{Decl} \bullet \text{CProc} \\
& \mid \diamond \text{Decl} \bullet \text{CProc} \\
& \mid [\text{CSExp} \; \text{Decl} \bullet \text{CProc} \\
& \mid ||| \; \text{Decl} \bullet \text{CProc} \\
& \mid \text{CProc} \setminus \text{CSExp} \\
& \mid \text{CProc} \Box \text{CProc} \\
& \mid \text{CProc} \setminus \text{CProc} \\
& \mid \text{CProc} \geq \text{CProc} \\
& \mid \text{CProc} \triangledown \text{CProc} \\
& \mid (\text{Decl} \bullet \text{ProcDef})(Exp^+) \\
& \mid \text{begin } PPar^* \\
& \quad \text{state } \text{SchemaExp} PPar^* \\
& \quad \bullet \text{CAction} \\
& \quad \text{end} \\
& \mid \text{begin } PPar^* \\
& \quad \bullet \text{CAction} \\
& \quad \text{end} 
\end{aligned}

\section*{A.6 Process paragraphs \textemdash PPar}

\begin{aligned} 
PPar &::= \quad ZPara \\
& \mid N \equiv \text{ParAction} \\
& \mid \text{NSExp} 
\end{aligned}

\section*{A.7 Parametrised Actions \textemdash ParAction}

\begin{aligned} 
ParAction &::= \quad \text{CAction} \\
& \mid \text{Decl} \bullet \text{ParAction} 
\end{aligned}
A.8 Circus Actions – CAction

\[
\begin{align*}
\text{CAction} & ::= (s) \\
& \mid \text{Name} \\
& \mid \text{Skip} \\
& \mid \text{Stop} \\
& \mid \text{Chaos} \\
& \mid \text{Comm} \rightarrow \text{CAction} \\
& \mid \text{CCommand} \\
& \mid (\text{Pred}) \& \text{CAction} \\
& \mid \text{CAction} ; \text{CAction} \\
& \mid \text{CAction} \& \text{CAction} \\
& \mid \text{CAction} \cap \text{CAction} \\
& \mid \text{CAction} \ NSExp \ | \text{CExp} \ | \text{NSExp} \ CAction \\
& \mid \text{CAction} \ | \text{CExp} \text{CAction} \\
& \mid \text{CAction} \ | \text{CAction} \\
& \mid \text{CAction} \ | \text{NSExp} \\
& \mid \text{CAction} \ | \text{CExp} \text{Exp}^+ \\
& \mid \text{CAction}[x/y, z/n] \\
& \mid \mu N \bullet \text{CAction} \\
& \mid (\text{Decl} \bullet \text{CAction})(\text{ZName}) \\
& \mid ; \text{Decl} \bullet \text{CAction} \\
& \mid \square \text{Decl} \bullet \text{CAction} \\
& \mid \lnot \text{Decl} \bullet \text{CAction} \\
& \mid \lbrack \text{CExp} \mid \text{Decl} \bullet \lbrack \text{NSExp} \rbrack \text{CAction} \\
& \mid \lbrack \text{CExp} \mid \text{Decl} \bullet \text{CAction} \\
& \mid \text{Decl} \bullet \lbrack \text{NSExp} \rbrack \text{CAction} \\
& \mid \text{Decl} \bullet \text{CAction}
\end{align*}
\]

A.9 Circus Communication

\[
\text{Comm} ::= \text{NCPParameter}\ast \\
\mid N[\text{Exp}^+]\text{CParameter}\ast
\]

A.10 Circus Communication – CParameter

\[
\begin{align*}
\text{CParameter} & ::= ?N \\
& \mid ?N : \text{Pred} \\
& \mid !\text{Exp} \\
& \mid .\text{Exp}
\end{align*}
\]
A.11  Circus Commands – \texttt{CCommand}

\[
\text{CCommand} ::= \quad N^+ := \text{Exp}^+ \\
| \quad \text{if } \text{GActions} \text{ fi} \\
| \quad \text{var Decl } \bullet \text{ CAction} \\
| \quad \text{val Decl } \bullet \text{ CAction} \\
| \quad \text{res Decl } \bullet \text{ CAction} \\
| \quad \text{vres Decl } \bullet \text{ CAction} \\
| \quad N^+ : [\text{Pred}, \text{Pred}] \\
| \quad N^+ : [\text{Pred}] \\
| \quad :[\text{Pred}, \text{Pred}] \\
| \quad :[\text{Pred}] \\
| \quad \{ \text{Pred} \} \\
| \quad [\text{Pred}]
\]

A.12  Circus Guards – \texttt{CGActions}

\[
\text{CGActions} ::= \quad \text{Pred } \rightarrow \text{CAction} \\
| \quad \text{CGActions } \parallel \text{ GActions}
\]

A.13  Circus Renaming – \texttt{CReplace}

\[
\text{CCommand} ::= \quad \text{CAction}[N^+/M^+] \\
| \quad \text{if } \text{CAction}[N^+:M^+]
\]
Appendix B

Omega Mapping Functions

B.1 Circus and Z Paragraphs

\[ \Omega_{ZPara}(\text{channel } CDecl) \triangleq \text{channel } CDecl \]
\[ \Omega_{ZPara}(\text{channelset } N \Rightarrow CExp) \triangleq \text{channelset } N \Rightarrow CExp \]
\[ \Omega_{ZPara}(Pr) \triangleq \Omega_{ProcDecl}(Pr) \]

B.2 Mapping Circus Processes Declaration

B.2.1 Circus Process – ProcDecl

\[ \Omega_{ProcDecl}(\text{process } N \triangleright P) \triangleq \text{process } N \triangleright \Omega_{ProcDecl}(P) \]
\[ \Omega_{ProcDecl}(\text{process } N(N^+) \triangleright P) \triangleq \text{process } N(N^+) \triangleright \Omega_{ProcDecl}(P) \]
\[ \Omega_{ProcDecl}(\text{process } N[N^+] \triangleright P) \triangleq \text{process } N[N^+] \triangleright \Omega_{ProcDecl}(P) \]

B.2.2 Circus Process – ProcessDef

\[ \Omega_{ProcessDef}(Decl \bullet PD) \triangleq (Decl \bullet \Omega_{ProcessDef}(PD)) \]
\[ \Omega_{ProcessDef}(Decl \odot PD) \triangleq Decl \odot \Omega_{ProcessDef}(PD) \]
\[ \Omega_{ProcessDef}(Proc) \triangleq \Omega_{CProc}(Proc) \]
B.3 Mapping Parameterised Circus Actions

\[\Omega_{\text{PPar}}(Z\text{Para}) \equiv \Omega_{\text{ZSchema}}(Z\text{Para})\]
\[\Omega_{\text{PPar}}(N \equiv PA) \equiv N \equiv\]
\[\Omega_{\text{ParAction}}(PA)\]
\[\Omega_{\text{PPar}}(NS\text{Exp}) \equiv NS\text{Exp}\]

B.3.1 Parametrised Actions – ParAction

\[\Omega_{\text{ParAction}}(\text{Decl} \bullet PA) \equiv \text{Decl} \bullet \Omega_{\text{ParAction}}(PA)\]
\[\Omega_{\text{ParAction}}(CA) \equiv \Omega_{\text{CAction}}(CA)\]

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

\(\Omega_A 1\)

\[\Omega_A(\text{Skip}) \equiv \text{Skip}\]
\[\Omega_A(\text{Stop}) \equiv \text{Stop}\]
\[\Omega_A(\text{Chaos}) \equiv \text{Chaos}\]

\(\Omega_A 2\)

\[\Omega_A(c \rightarrow A) \equiv c \rightarrow \Omega_A(A)\]

\(\Omega_A 3\)

\[\Omega_A(c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A) \equiv\]
\[\begin{align*}
\text{mget}.v_0?v_0 & \rightarrow \ldots \rightarrow \text{mget}.v_n?v_n \rightarrow \\
\text{mget}.l_0?v_0 & \rightarrow \ldots \rightarrow \text{mget}.l_m?v_m \rightarrow \\
c.e(v_0, \ldots, v_n, v_0, \ldots, v_m) & \rightarrow \Omega_A(A)
\end{align*}\]
where

\[ FV(e) = (v_0, \ldots, v_n, l_0, \ldots, l_m) \]

**Ω₄**

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 \( v_n \in \text{StateVariables} \), we have the following translation rule.

\[ Ω₄(c?v_n → A) ≡ (c?vv_n → mset.v_n!(v_0, \ldots, vv_n, vl_0, \ldots, vl_m) → Ω₄(A)) \]

where

\[ v_n ∈ (v_0, \ldots, v_n, l_0, \ldots, l_m) \]

**Ω₅**

\[ Ω₅(c?x : P(x, v_0, \ldots, v_n, l_0, \ldots, l_m) → A) ≡ mget.v_0?v_0 → \ldots → mget.v_n?vv_n → mget.l_0?vl_0 → \ldots → mget.l_m?vl_m → c?x : P(x, v_0, \ldots, vv_n, vl_0, \ldots, vl_m) → Ω₅(A) \]

where

\[ x ∈ \text{wrtV}(A) \]

**Ω₆**

\[ Ω₆(c?x → A) ≡ c?x → Ω₆(A) \]

**Ω₇**

\[ Ω₇(A₁ || A₂) ≡ Ω₇(A₁) || Ω₇(A₂) \]

**Ω₈**

\[ Ω₈(c!e(v_0, \ldots, v_n, l_0, \ldots, l_m) → A) ≡ c!e(v_0, \ldots, v_n, l_0, \ldots, l_m) → A \]

**Ω₉**

\[ Ω₉(g(v_0, \ldots, v_n, l_0, \ldots, l_m) → A) ≡ mget.v_0?vv_0 → \ldots → mget.v_n?vv_n → mget.l_0?vl_0 → \ldots → mget.l_m?vl_m → g(v_0, \ldots, vv_n, vl_0, \ldots, vl_m) & Ω₉(A) \]

**Ω₁₀**

\[ Ω₁₀(A₁ ; A₂) ≡ Ω₁₀(A₁) ; Ω₁₀(A₂) \]

**Ω₁₁**

\[ Ω₁₁(A₁ ⊓ A₂) ≡ Ω₁₁(A₁) ⊓ Ω₁₁(A₂) \]

**Ω₁₂**

\[ Ω₁₂(A₁ □ A₂) ≡ \left( mget.v_0?vv_0 → \ldots → mget.v_n?vv_n → mget.l_0?vl_0 → \ldots → mget.l_m?vl_m → (Ω′₅(A₁) □ Ω′₅(A₂)) \right) \]
The definition of parallel composition (and interleaving), as defined in the D24.1, has a MemoryMerge, MRG and Merge components and channel sets. Whilst translating them into CSP, the tool would rather expand their definition.

\[ \Omega_A(A1 \parallel ns1 \mid cs \mid ns2 \parallel A2) \equiv \Gamma_A(A1 \mid ns1 \mid cs \mid ns2 \parallel A2) \]

\[ \Omega_A(\text{x} : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Omega_A(A(v_1) ; \ldots ; A(v_n)) \]

\[ \Omega_A(\square(x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Omega_A(A(v_1) \square \ldots \square A(v_n)) \]

\[ \Omega_A(\cap(x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Omega_A(A(v_1) \cap \ldots \cap A(v_n)) \]

\[ \Omega_A(\{cs\} x : (v_1, \ldots, v_n) \bullet \{ns(x)\} A(x)) \equiv \]

\[ \left( \Omega_A(A(v_1)) \right) \left( \left( \Omega_A(A(v_n - 1)) \right) \left( \ldots \left( \Omega_A(A(v_n)) \right) \right) \right) \]

\[ \Omega_A(\text{val Decl} \bullet P) \equiv \text{val Decl} \bullet \Omega_A(P) \]

\[ \Omega_A(x_0, \ldots, x_n := c_0 \left( v_0, \ldots, v_n, \right) l_0, \ldots, l_m, \ldots, c_n \left( v_0, \ldots, v_n, \right) l_0, \ldots, l_m \) \equiv \]

\[ mget.v_0 ? v_0 \rightarrow \ldots \rightarrow mget.v_n ? v_n \rightarrow \]

\[ mget.l_0 ? v_0 \rightarrow \ldots \rightarrow mget.l_m ? v_m \rightarrow \]

\[ mset.x_0 ! e_0 (v_0, \ldots, v_n, v_0, \ldots, v_m) \rightarrow \ldots \rightarrow \]

\[ mset.x_n ! e_n (v_0, \ldots, v_n, v_0, \ldots, v_m) \rightarrow \text{Skip} \]

\[ \Omega_A(A \setminus cs) \equiv \Omega_A(A) \setminus cs \]

\[ \Omega_A\left( \left( \begin{array}{c} \text{if } g_0(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow A_0 \\
\ldots \\
\text{fi} \\
\end{array} \right) \left( \begin{array}{c} \text{if } g_n(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow \Omega_A(A_n) \\\n\ldots \\
\text{fi} \\
\end{array} \right) \right) \equiv \]

\[ \Omega_A(\mu X \bullet A(X)) \equiv \mu X \bullet \Omega_A(A(X)) \]
ΩA 23

\[
\Omega_A(x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \left\langle \begin{array}{c}
A(v_1) \\
\{ \text{ns}(v_1) \} \cup \{ x : (v_2, \ldots, v_n) \bullet \text{ns}(x) \} \\
\quad \vdots \\
\quad \{ \text{ns}(v_n - 1) \} \cup \{ \text{ns}(v_n) \} \\
\end{array} \right]
\]

ΩA 24

\[
\Omega_A(A[old_1, \ldots, old_n := new_1, \ldots, new_n]) \equiv A[new_1, \ldots, new_n/old_1, \ldots, old_n]
\]

B.3.3 Definitions of \( \Omega_A' \)

ΩA' 1

\( \Omega_A'(\text{Skip}) \equiv \text{Skip} \)
\( \Omega_A'(\text{Stop}) \equiv \text{Stop} \)
\( \Omega_A'(\text{Chaos}) \equiv \text{Chaos} \)

ΩA' 2

\( \Omega_A'(c \rightarrow A) \equiv c \rightarrow \Omega_A'(A) \)

ΩA' 3

\( \Omega_A'(c.c \rightarrow A) \equiv c(v_0, \ldots, v_n, v_l_0, \ldots, v_l_m) \rightarrow \Omega_A'(A) \)

ΩA' 4

\( \Omega_A'(c!e \rightarrow A) \equiv c!e \rightarrow A \)

ΩA' 5

\( \Omega_A'(g \rightarrow A) \equiv g \rightarrow \Omega_A'(A) \)

ΩA' 6

\( \Omega_A'(c?x \rightarrow A) \equiv c?x \rightarrow \Omega_A'(A) \)

ΩA' 7

\( \Omega_A'(c?x : P \rightarrow A) \equiv c?x : P \rightarrow \Omega_A'(A) \)

where

\( x \in \text{wrt}V(A) \)

ΩA' 8

\( \Omega_A'(A_1; A_2) \equiv \Omega_A'(A_1); \Omega_A'(A_2) \)

ΩA' 9

\( \Omega_A'(A_1 \cap A_2) \equiv \Omega_A'(A_1) \cap \Omega_A'(A_2) \)
\( \Omega'_A \) 10

\( \Omega'_A (A_1 \sqcap A_2) \equiv (\Omega'_A (A_1) \sqcap \Omega'_A (A_2)) \)

\( \Omega'_A \) 11

\( \Omega'_A (A_1 \parallel ns1 | cs | ns2 \parallel A_2) \equiv \Gamma_A (A_1 \parallel ns1 | cs | ns2 \parallel A_2) \)

\( \Omega'_A \) 12

\( \Omega'_A (\bullet x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Omega'_A (A(v_1); \ldots; A(v_n)) \)

\( \Omega'_A \) 13

\( \Omega'_A (\Box x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Omega'_A (A(v_1) \Box \ldots \Box A(v_n)) \)

\( \Omega'_A \) 14

\( \Omega'_A (\cap x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Omega'_A (A(v_1) \cap \ldots \cap A(v_n)) \)

\( \Omega'_A \) 15

\( \Omega'_A (\parallel cs \parallel x : (v_1, \ldots, v_n) \bullet \parallel ns(x) \parallel A(x)) \equiv \)

\[
\begin{pmatrix}
\Omega'_A (A(v_1)) \\
\ldots \\
\Omega'_A (A(v_n)) \\
\end{pmatrix}
\]

\( \Omega'_A \) 16

\( \Omega'_A (\text{val Decl} \bullet P) \equiv \text{val Decl} \bullet \Omega'_A (P) \)

\( \Omega'_A \) 17

\( \Omega_A \left( x_0, \ldots, x_n := e_0, \ldots, e_n \right) \equiv \text{mset}. x_0 ! e_0 \rightarrow \ldots \rightarrow \text{mset}. x_n ! e_n \rightarrow \text{Skip} \)

\( \Omega'_A \) 18

\( \Omega'_A (A \setminus cs) \equiv \Omega'_A (A) \setminus cs \)

\( \Omega'_A \) 19

\( \Omega'_A \left( \begin{array}{c}
\text{if } g_0 \rightarrow A_0 \\
\vdots \\
\text{if } g_n \rightarrow A_n \\
\end{array} \right) \equiv \left( \begin{array}{c}
\text{if } g_0 \rightarrow \Omega'_A (A_0) \\
\vdots \\
\text{if } g_n \rightarrow \Omega'_A (A_n) \\
\end{array} \right) \)

\( \Omega'_A \) 20

\( \Omega'_A (\mu X \bullet A(X)) \equiv \mu X \bullet \Omega'_A (A(X)) \)

\( \Omega'_A \) 21

\( \Omega'_A (\parallel x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \)

\[
\begin{pmatrix}
\Omega'_A (A(v_1)) \\
\ldots \\
\Omega'_A (A(v_n)) \\
\end{pmatrix}
\]

\( \Omega'_A \) 22

\( \Omega'_A (A[old_1, \ldots, old_n := new_1, \ldots, new_n]) \equiv A[new_1, \ldots, new_n / old_1, \ldots, old_n] \)
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**

\[ \begin{align*}
\Gamma_A(\text{Skip}) & \equiv \text{Skip} \\
\Gamma_A(\text{Stop}) & \equiv \text{Stop} \\
\Gamma_A(\text{Chaos}) & \equiv \text{Chaos}
\end{align*} \]

\[ \begin{align*}
\Gamma_A(c \to A) & \equiv c \to \Gamma_A(A)
\end{align*} \]

\[ \begin{align*}
\Gamma_A(c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) & \equiv \\
& \left(\begin{array}{c}
\text{lget.} v_0? v_0 \to \ldots \to \text{lget.} v_n? v_n \\
\text{lget.} l_0? v_0 \to \ldots \to \text{lget.} l_m? v_m \\
c.e(v_0, \ldots, v_n, vl_0, \ldots, vl_m) \to \Gamma'_A(A)
\end{array}\right)
\end{align*} \]

where \( FV(e) = (v_0, \ldots, v_n, l_0, \ldots, l_m) \)

\[ \begin{align*}
\Gamma_A(c!e(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) & \equiv c.e(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A
\end{align*} \]

\[ \begin{align*}
\Gamma_A(g(v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) & \equiv \\
& \left(\begin{array}{c}
\text{lget.} v_0? v_0 \to \ldots \to \text{lget.} v_n? v_n \\
\text{lget.} l_0? v_0 \to \ldots \to \text{lget.} l_m? v_m \\
g(v_0, \ldots, v_n, vl_0, \ldots, vl_m) & \& \Gamma'_A(A)
\end{array}\right)
\end{align*} \]

\[ \begin{align*}
\Gamma_A(c?q x : P(x, v_0, \ldots, v_n, l_0, \ldots, l_m) \to A) & \equiv \\
& \left(\begin{array}{c}
\text{lget.} v_0? v_0 \to \ldots \to \text{lget.} v_n? v_n \\
\text{lget.} l_0? v_0 \to \ldots \to \text{lget.} l_m? v_m \\
c?q x : P(x, v_0, \ldots, v_n, vl_0, \ldots, vl_m) \to \Gamma'_A(A)
\end{array}\right)
\end{align*} \]

where \( x \in wrtV(A) \)

\[ \begin{align*}
\Gamma_A(A_1 ; A_2) & \equiv \Gamma_A(A_1) ; \Gamma_A(A_2)
\end{align*} \]

\[ \begin{align*}
\Gamma_A(A_1 \cap A_2) & \equiv \Gamma_A(A_1) \cap \Gamma_A(A_2)
\end{align*} \]
The definition of parallel composition (and interleaving), as defined in the D24.1, has a MemoryMerge, MRGI and Merge components and channel sets. Whilst translating them into CSP, the tool would rather expand their definition.

\[ \Gamma_A( ; \ x : (v_1, ..., v_n) \bullet A(x)) \equiv \Gamma_A(A(v_1); \ldots; A(v_n)) \]

\[ \Gamma_A(\square x : \langle v_1, ..., v_n \rangle \bullet A(x)) \equiv \Gamma_A(A(v_1) \square \ldots \square A(v_n)) \]

\[ \Gamma_A(\cap x : \langle v_1, ..., v_n \rangle \bullet A(x)) \equiv \Gamma_A(A(v_1) \cap \ldots \cap A(v_n)) \]

\[ \Gamma_A([\text{cs}] x : \langle v_1, ..., v_n \rangle \bullet [\text{ns}(x)] A(x)) \equiv \]

\[ \begin{align*}
\langle \text{ns}(v_1) & \mid \text{cs} \mid \bigcup \{ \langle x : \{v_2, ..., v_n\} \bullet \text{ns}(x)\} \bigg] \\
& \quad \ldots \\
& \quad \left( \Gamma_A(A(v_n - 1)) \right) \\
& \quad \left( \text{ns}(v_n - 1) \mid \text{cs} \mid \text{ns}(v_n) \right)
\end{align*} \]

\[ \Gamma_A(\text{val Decl} \bullet P) \equiv \text{val Decl} \bullet \Gamma_A(P) \]

\[ \Gamma_A( x_0, \ldots, x_n := e_0(v_0, \ldots, v_n), \ldots, e_n(v_0, \ldots, v_n) \bullet l_0, \ldots, l_m) \equiv \]

\[ \begin{align*}
& l\text{get}.v_0?v_0 \rightarrow \ldots \rightarrow l\text{get}.v_n?v_n \rightarrow \\
& l\text{get}.l_0?l_0 \rightarrow \ldots \rightarrow l\text{get}.l_m?l_m \rightarrow \\
& l\text{set}.x_0!e_0(v_0, \ldots, v_n, l_0, \ldots, l_m) \\
& l\text{dots} \\
& l\text{set}.x_0!e_n(v_0, \ldots, v_n, l_0, \ldots, l_m) \rightarrow \text{Skip}
\end{align*} \]

\[ \Gamma_A(A \setminus \text{cs}) \equiv \Gamma_A(A) \setminus \text{cs} \]
\[ \Gamma_A (\mu X \cdot A(X)) \cong \mu X \cdot \Gamma_A (A(X)) \]

\[ \Gamma_A (A[old_1, ..., old_n := new_1, ..., new_n]) \cong A[new_1, ..., new_n/old_1, ..., old_n] \]

\[ \text{B.4.2 Definitions of } \Gamma'_A \]

\[ \Gamma'_A (\text{Skip}) \cong \text{Skip} \]
\[ \Gamma'_A (\text{Stop}) \cong \text{Stop} \]
\[ \Gamma'_A (\text{Chaos}) \cong \text{Chaos} \]

\[ \Gamma'_A (c \rightarrow A) \cong c \rightarrow \Gamma'_A (A) \]

\[ \Gamma'_A (c.e \rightarrow A) \cong c(e_0, ..., e_n, v_0, ..., v_m) \rightarrow \Gamma'_A (A) \]

\[ \Gamma'_A (c!c \rightarrow A) \cong c.e \rightarrow A \]

\[ \Gamma'_A (c?x \rightarrow A) \cong c?x \rightarrow \Gamma'_A (A) \]

\[ \Gamma'_A (c?x : P \rightarrow A) \cong c?x : P \rightarrow \Gamma'_A (A) \]
where
\[ x \in \text{wrt} V(A) \]

\[ \Gamma'_A \ 7 \]

\[ \Gamma'_A(g \& A) \equiv g \& \Gamma'_A(A) \]

\[ \Gamma'_A \ 8 \]

\[ \Gamma'_A(A_1 \mid A_2) \equiv \Gamma'_A(A_1) \mid \Gamma_A(A_2) \]

\[ \Gamma'_A \ 9 \]

\[ \Gamma'_A(A_1 \cap A_2) \equiv \Gamma'_A(A_1) \cap \Gamma'_A(A_2) \]

\[ \Gamma'_A \ 10 \]

\[ \Gamma'_A(A_1 \sqcap A_2) \equiv (\Gamma'_A(A_1) \sqcap \Gamma'_A(A_2)) \]

\[ \Gamma'_A \ 11 \]

\[ \Gamma'_A(A_1 \parallel ns_1 \mid cs \mid ns_2 \parallel A_2) \equiv \]

\[ \left( \begin{array}{c}
\text{lget}\ A_n \parallel v_n \rightarrow \ldots \rightarrow \text{lget}\ A_n \parallel v_{n_m} \rightarrow \\
\text{lget}\ A_n \parallel v_0 \rightarrow \ldots \rightarrow \text{lget}\ A_n \parallel v_{m_n} \rightarrow \\
\begin{array}{c}
\left( \Gamma'_A(A_1) \mid \text{terminate} \rightarrow \text{Skip} \right) \\
\left( \{ \{ \} | \text{MEM}_l | \{ \} \} | \text{MEM}_r \right) \\
\{ \{ \} | \text{MEM}_l | \{ \} \} | \text{MEM}_r \\
\{ \} | \text{MEM}_l | \{ \} \\
\text{MemoryMerge}(\{ v_0 \rightarrow v_{m_n}, \ldots \}, \text{LEFT}) \\
\text{MemoryMerge}(\{ v_0 \rightarrow v_{m_n}, \ldots \}, \text{RIGHT}) \\
\text{Merge} \quad \# \text{mleft, mright} \end{array}
\end{array} \right) \]

\[ \Gamma'_A \ 12 \]

\[ \Gamma'_A(; \ x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Gamma'_A(A(v_1) ; \ldots ; A(v_n)) \]

\[ \Gamma'_A \ 13 \]

\[ \Gamma'_A(\square x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Gamma'_A(A(v_1) \square \ldots \square A(v_n)) \]

\[ \Gamma'_A \ 14 \]

\[ \Gamma'_A(\sqcap x : (v_1, \ldots, v_n) \bullet A(x)) \equiv \Gamma'_A(A(v_1) \sqcap \ldots \sqcap A(v_n)) \]

\[ \Gamma'_A \ 15 \]

\[ \Gamma'_A([\text{cs}] x : (v_1, \ldots, v_n) \bullet [\text{ns}(x)] A(x)) \equiv \]

\[ \left( \begin{array}{c}
\text{ns}(v_1) \parallel \text{cs} | \bigcup \{ x : (v_2, \ldots, v_n) \bullet \text{ns}(x) \} \\
\Gamma'_A(A(v_n - 1)) \\
\ldots \\
\text{ns}(v_n - 1) \parallel \text{cs} | \text{ns}(v_n) \\
\text{ns}(v_1) \parallel \text{cs} | \bigcup \{ x : (v_2, \ldots, v_n) \bullet \text{ns}(x) \} \end{array} \right) \]

\[ \Gamma'_A \ 16 \]

\[ \Gamma'_A(\text{val Decl} \bullet P) \equiv \text{val Decl} \bullet \Gamma'_A(P) \]
\( \Gamma_A' 17 \)
\[
\Gamma_A' (x_0, \ldots, x_n := e_0, \ldots, e_n) \equiv \left( \begin{array}{l}
\text{mset} \cdot x_0! e_0 \rightarrow \\
\vdots \rightarrow \\
\text{mset} \cdot x_n! e_n \rightarrow \text{Skip}
\end{array} \right)
\]

\( \Gamma_A' 18 \)
\[
\Gamma_A'(A \setminus cs) \equiv \Gamma_A'(A) \setminus cs
\]

\( \Gamma_A' 19 \)
\[
\Gamma_A' \left( \begin{array}{c}
\text{if } g_0 \rightarrow A_0 \\
\vdots \\
\text{if } g_n \rightarrow A_n
\end{array} \right) \equiv \left( \begin{array}{c}
\text{if } g_0 \rightarrow \Gamma_A'(A_0) \\
\vdots \\
\text{if } g_n \rightarrow \Gamma_A'(A_n)
\end{array} \right)
\]

\( \Gamma_A' 20 \)
\[
\Gamma_A'(\mu X \cdot A(X)) \equiv \mu X \cdot \Gamma_A'(A(X))
\]

\( \Gamma_A' 21 \)
\[
\Gamma_A'(\left\{ x: (v_1, \ldots, v_n) \cdot A(x) \right\) \equiv 
\left( \begin{array}{c}
A(v_1) \\
\left\{ \text{ns} \cdot \{ x: (v_2, \ldots, v_n) \cdot \text{ns}(x) \} \right\} \\
\vdots \\
\left\{ \text{ns} \cdot \{ v_n \cdot \text{ns}(v_n) \} \right\} \\
A(v_n)
\end{array} \right)
\]

\( \Gamma_A' 22 \)
\[
\Gamma_A'(A[old_1, \ldots, old_n := new_1, \ldots, new_n]) \equiv A[new_1, \ldots, new_n/old_1, \ldots, old_n]
\]
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 \( \text{UNIVERSE} \) type is defined as follows.

\[
\Upsilon_{\text{CircParagraphs}}([ \text{UNIVERSE} ]) \equiv \\
\text{datatype \text{UNIVERSE} } = \text{MK}_{\text{Universe}}(\text{ulst}) \\
\text{MK}_\text{subtype}(\text{fulst}) \\
\text{MK}_\text{value}(\text{fulst}) \\
\text{MK}_\text{type}(\text{fulst})
\]

provided

- \( \delta(\emptyset) \not\in \text{spec} \) – There is at least one element in the \( \delta \) mapping.
- \( \text{spec} \) is the whole specification
- \( \text{ulst} = \text{DEF}_{\text{Universe}}(\text{spec}) \) – list of mappings from variables to types, from \( \delta \) function
- \( \text{fulst} = \text{GET}_{\text{Types}}(\text{spec}) \) – list of all types from \( \delta \) function

Then, we define the translation for the channels and sets of the Memory model.

\[
\Upsilon_{\text{CircParagraphs}}(\text{channel } mget, mset : \text{NAME} \times \text{UNIVERSE}) \equiv \\
\Upsilon_{\text{CDecl}}(\text{channel } mget, mset : \text{NAME} \times \text{UNIVERSE}) \\
\Upsilon_{\text{CDecl}}(\text{channel } \text{terminate}) \equiv \\
\Upsilon_{\text{CDecl}}(\text{channel } \text{terminate}) \\
\Upsilon_{\text{CDecl}}(\text{channelset } \text{MEMI} == \{ mget, mset, \text{terminate } \}) \equiv \\
\text{MEM}_I = \Upsilon_{\text{CSExp}}(\{ mget, mset, \text{terminate } \}) \\
\Upsilon_{\text{CDecl}}(\text{BINDING} \equiv \text{NAME} \times \{ \}) \equiv \\
\text{BINDINGS}_{T} = (\text{set}(b) | b \leftarrow \text{set}(\text{distCartProd}(\text{VALUES}_{T}))) \\
\ldots \\
\text{BINDINGS}_{T_n} = (\text{set}(b) | b \leftarrow \text{set}(\text{distCartProd}(\text{VALUES}_{T_n})))
\]

Finally, we define the translation of the remainder constructs.

\[
\Upsilon_{\text{CDecl}}(N ::= a_1 | \ldots | a_n) \equiv \text{datatype } N = \Upsilon_{\text{ZBranch}}(a_1 | \ldots | a_n) \\
\Upsilon_{\text{CDecl}}(P) \equiv \Upsilon_{\text{ProcDecl}}(P) \\
\Upsilon_{\text{CDecl}}(\text{channel } c) \equiv \Upsilon_{\text{CDecl}}(c) \\
\Upsilon_{\text{CDecl}}(\text{channelset } \text{CN} == \text{CS}) \equiv \text{CN} = \Upsilon_{\text{CSExp}}(\text{CS}) \\
\Upsilon_{\text{CDecl}}(N ::= \text{expr}) \equiv N = \Upsilon_{\text{ZExpr}}
\]
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.

\[
\begin{align*}
\Upsilon_{CDec}(\text{channel } a) & \triangleq \text{channel } a \\
\Upsilon_{CDec}(\text{channel } a : T) & \triangleq \text{channel } a : T
\end{align*}
\]

A channel declaration can be either of form \text{CChan} or \text{CChanDecl}. For \text{CChan}, we can have \text{channel terminate}, whereas for \text{CChanDecl}, \text{channel mget : NAME } \times \text{BINDING}. Thus, we filter both cases into \(x_1\) for \text{CChan} and \(x_2\) for \text{CChanDecl}, and then, display them accordingly.

C.1.2 Mapping Function for Channel Set Expressions

\[
\begin{align*}
\Upsilon_{CSExp}(\{|xs\}|) & \triangleq \{ \Upsilon_{CSExp}(\text{get}_{cs}(xs)) \} \\
\Upsilon_{CSExp}(CS_1 \setminus CS_2) & \triangleq \text{diff}(\Upsilon_{CSExp}(CS_1), \Upsilon_{CSExp}(CS_2)) \\
\Upsilon_{CSExp}(CS_1 \cup CS_2) & \triangleq \text{union}(\Upsilon_{CSExp}(CS_1), \Upsilon_{CSExp}(CS_2)) \\
\Upsilon_{CSExp}(CS_1 \cap CS_2) & \triangleq \text{inter}(\Upsilon_{CSExp}(CS_1), \Upsilon_{CSExp}(CS_2)) \\
\Upsilon_{CSExp}(| |) & \triangleq (\) \\
\Upsilon_{CSExp}(CS) & \triangleq CS
\end{align*}
\]

provided

- \(CS, CS_1\) and \(CS_2\) are channel sets, \(CSExp\).

C.1.3 Mapping Circus Processes Declaration

This is the translation rules for \text{ProcDecl}. Up to the date, we don’t have a translation rule for generic processes.

\[
\Upsilon_{ProcDecl}(\text{process } P) \triangleq \Upsilon_{ProcDef}(P) \triangleq P\Upsilon_{ProcessDef}(PD)
\]

provided \(P\) is the name of a Circus process.

C.1.4 Mapping Circus Processes Definition

\[
\begin{align*}
\Upsilon_{ProcessDef}(Proc) & \triangleq \Upsilon_{CProc}(Proc) \\
\Upsilon_{ProcessDef}(Decl \bullet Proc) & \triangleq (\Upsilon_{GenFilt}_\text{list}) = \Upsilon_{CProc}(Proc)
\end{align*}
\]

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 \( \text{CGenProc} \left( N [\text{Exp}^+] \right) \), \( \text{CSimpIndexProc} \), and \( \text{CParamProc} \left( N \left( \text{Exp}^+ \right) \right) \) are not yet implemented.

\[
\begin{align*}
\Upsilon \text{CProc} (P1 \boxdot P2) & \equiv \Upsilon \text{GProc} (P1) \cup \Upsilon \text{GProc} (P2) \\
\Upsilon \text{CProc} (P1 \setminus CS) & \equiv \Upsilon \text{CProc} (P1) \setminus \Upsilon \text{PredCS} (CS) \\
\Upsilon \text{CProc} (P1 \triangledown P2) & \equiv \Upsilon \text{CProc} (P1) \setminus \Upsilon \text{CProc} (P2) \\
\Upsilon \text{CProc} (P1 \parallel P2) & \equiv \Upsilon \text{GProc} (P1) \parallel \Upsilon \text{GProc} (P2) \\
\Upsilon \text{CProc} (P) & \equiv P \\
\Upsilon \text{CProc} (P1 \parallel CS \parallel P2) & \equiv \Upsilon \text{GProc} (P1) \parallel \Upsilon \text{PredCS} (CS) \parallel \Upsilon \text{GProc} (P2) \\
\Upsilon \text{CProc} (\Box x : S \bullet P1) & \equiv \{ \} \times \Upsilon ZExpr (S) \parallel \Upsilon \text{CProc} (P2) \\
\Upsilon \text{CProc} (\Box x : S \bullet P1) & \equiv \{ \} \times \Upsilon ZExpr (S) \parallel \Upsilon \text{CProc} (P2) \\
\Upsilon \text{CProc} (\Box[CS] x : S \bullet P1) & \equiv \{ \} \times \Upsilon ZExpr (S) \parallel \Upsilon \text{CProc} (P2) \\
\Upsilon \text{CProc} (\begin{align*}
(\begin{align*}
\text{begin} & AL \bullet MA \text{end} & \equiv \\
\text{let} & \Upsilon PPar \text{list} (AL) \\
\text{within} & \Upsilon \text{CAction} (MA)
\end{align*}
\end{align*}
) & \equiv \Upsilon \text{CAction} (MA) \\
\Upsilon \text{CProc} (\begin{align*}
(\begin{align*}
\text{begin} & \bullet MA \text{end} & \equiv \\
\text{let} & \Upsilon PPar \text{list} (AL) \\
\text{within} & \Upsilon \text{CAction} (MA)
\end{align*}
\end{align*}
) & \equiv \Upsilon \text{CAction} (MA) \\
\Upsilon \text{CProc} (\text{Proc}[NL := EL]) & \equiv P \{ \Upsilon \text{Rename} (NL, EL) \}
\end{align*}
\]

provided

\[
\begin{align*}
\bullet P & \text{ is a process name.} \\
\bullet P1 \text{ and } P2 & \text{ is a process } \text{CProc} \\
\bullet cs & \text{ is a channel set } \text{CSExp} \\
\bullet MA & \text{ is the main action of the } \text{Circus} & \text{ process.} \\
\bullet AL & \text{ is the list of actions.}
\end{align*}
\]

This function maps any renaming, to its equivalent syntax in \( \text{CSP}_M \).

\[
\begin{align*}
\Upsilon \text{Rename} (x \# xs, y \# xs) & \equiv \Upsilon \text{Comm} (x) \left< - \Upsilon \text{Comm} (y), \Upsilon \text{Rename} (xs, xs) \\
\Upsilon \text{Rename} ([x], [y] =) & \equiv \Upsilon \text{Comm} (x) \left< - \Upsilon \text{Comm} (y)
\end{align*}
\]

C.1.6 Mapping Circus Processes Paragraphs

\[
\begin{align*}
\Upsilon \text{PPar} (P \equiv \text{Decl} \bullet A) & \equiv P \{ \Upsilon \text{ZGenFilt} \text{list} (\text{Decl}) \} = \Upsilon \text{CAction} (A) \\
\Upsilon \text{PPar} (P \equiv A) & \equiv P = \Upsilon \text{CAction} (A)
\end{align*}
\]
C.1.7 Mapping Circus Actions

\[\begin{align*}
\mathcal{A}_A(c \to A) & \equiv c \to \mathcal{A}_A(A) \\
\mathcal{A}_A(c?x \to A) & \equiv c?x \to \mathcal{A}_A(A) \\
\mathcal{A}_A(c.v \to A) & \equiv c.v \to \mathcal{A}_A(A) \\
\mathcal{A}_A(c?x : P \to A) & \equiv c?x : \mathcal{A}_Z(\text{Exp})(P) \to \mathcal{A}_A(A) \\
\mathcal{A}_A(mget.x?v_x : (\text{type}(x)) \to \mathcal{A}_A(A) \\
\mathcal{A}_A(mset.x,v_x \to A) & \equiv \text{mset.x, (\text{type.value}(v_x)) \to \mathcal{A}_A(A)} \\
\mathcal{A}_A(A \& B) & \equiv \mathcal{A}_A(A) \cup \mathcal{A}_A(B) \\
\mathcal{A}_A(A \& B) & \equiv \mathcal{A}_A(A) \cup \mathcal{A}_A(B) \\
\mathcal{A}_A(g \& A) & \equiv \text{let } \mu A = \mathcal{A}_A(A) \text{ within } \mu A \\
\mathcal{A}_A(A \& B) & \equiv [\{] x : \mathcal{A}_P(S) \@ \mathcal{A}_A(A) \\
\mathcal{A}_A(A; B) & \equiv \mathcal{A}_A(A) ; \mathcal{A}_A(B) \\
\mathcal{A}(_{\text{Skip}}) & \equiv \text{SKIP} \\
\mathcal{A}(_{\text{Stop}}) & \equiv \text{STOP} \\
\mathcal{A}(_{\text{Chaos}}) & \equiv \text{CHAOSS}
\end{align*}\]

C.1.8 Mapping Circus Commands

Assignments are translated into prefixed actions with \textit{mget} and \textit{mset} with the \(\Omega\) functions. Therefore, we only have to translate conditional commands with if \(\text{if} - \text{then} - \text{else}\). Moreover, we do not have to provide any translation for local variable declaration \textit{var} as they are promoted to the \textit{Memory} state process.

\[\begin{align*}
\mathcal{A}_A\left(\begin{array}{c}
\text{if } g_0 \longrightarrow A_0 \\
\vdots \\
g_n \longrightarrow A_n
\end{array}\right) & \equiv \left(\begin{array}{c}
g_0 \to \mathcal{A}_A(A_0) \\
\vdots \\
g_n \to \mathcal{A}_A(A_n)
\end{array}\right)
\end{align*}\]

C.1.9 Mapping Circus Guarded Actions

\[\begin{align*}
\mathcal{A}_A\left(\begin{array}{c}
[\{] g_0 \& A_0 \\
\vdots \\
[\{] g_0 \& A_n
\end{array}\right) & \equiv \left(\begin{array}{c}
g_0 \& \mathcal{A}_A(A_0) \\
\vdots \\
g_n \& \mathcal{A}_A(A_n)
\end{array}\right)
\end{align*}\]

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 definitions and some may use the auxiliary definitions in \textit{CSP}_M included in the \textit{functions_aux.csp} and \textit{sequences_aux.csp}. 

119
C.2.1 Mapping Function for Expressions

\[ Z_{Expr}(a) = \begin{cases} 
\text{NatValue} & m \in \mathbb{Z} \\
\text{value(def}_\_\text{prefix } T)(v_a) & a_T \in \text{UNIVERSE} \\
\text{a} & a \notin \text{State} 
\end{cases} \]

\[ (\text{dom}(\text{ZExpr}(a))) = \text{ran}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(m \mod n)) = \text{diff}(\text{ZExpr}(m), \text{ZExpr}(n)) \]

\[ (\text{ZExpr}(m \setminus n)) = \text{dres}(\text{ZExpr}(m), \text{ZExpr}(n)) \]

\[ (\text{ZExpr}(m \div n)) = \text{comp}(\text{ZExpr}(m), \text{ZExpr}(n)) \]

\[ (\text{ZExpr}(m \div n)) = \text{rres}(\text{ZExpr}(m), \text{ZExpr}(n)) \]

\[ (\text{ZExpr}(m \div n)) = \text{pfun}(\text{ZExpr}(m), \text{ZExpr}(n)) \]

\[ (\text{ZExpr}(m \div n)) = -\text{ZExpr}(n) \]

\[ (\text{ZExpr}(m \div n)) = \text{over}(\text{ZExpr}(a), \text{ZExpr}(b), \text{ZExpr}(c)) \]

\[ (\text{ZExpr}(a \oplus b)) = \text{oplus}(\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Set}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Seq}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{diff}(\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = (\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \{} \]

\[ (\text{ZExpr}(a \mapsto b)) = \{a, \ldots, c\} \]

\[ (\text{ZExpr}(a \mapsto b)) = <b_{-a}> \]

\[ (\text{ZExpr}(a \mapsto b)) = <sv_{-a}> \]

\[ (\text{ZExpr}(a \mapsto b)) = <y | y < - ns> \]

\[ (\text{ZExpr}(a \mapsto b)) = <a, \ldots, c> \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Set}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Seq}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{diff}(\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = (\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \{} \]

\[ (\text{ZExpr}(a \mapsto b)) = \{a, \ldots, c\} \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Set}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Seq}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{diff}(\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = (\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \{} \]

\[ (\text{ZExpr}(a \mapsto b)) = \{a, \ldots, c\} \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Set}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{Seq}(\text{ZExpr}(a)) \]

\[ (\text{ZExpr}(a \mapsto b)) = \text{diff}(\text{ZExpr}(a), \text{ZExpr}(b)) \]

\[ (\text{ZExpr}(a \mapsto b)) = (\text{ZExpr}(a), \text{ZExpr}(b)) \]
C.2.2 Mapping Functions for Predicates

\[ Y_{ZPred}(\text{if } x \text{ then } y \text{ else } z) \equiv \text{if } Y_{ZPred}(x) \text{then } Y_{ZPred}(y) \text{else } Y_{ZPred}(z) \]
\[ Y_{ZPred}(a \geq b) \equiv Y_{ZExpr}(a) \geq Y_{ZPred}(b) \]
\[ Y_{ZPred}(a > b) \equiv Y_{ZExpr}(a) > Y_{ZPred}(b) \]
\[ Y_{ZPred}(a \leq b) \equiv Y_{ZExpr}(a) \leq Y_{ZPred}(b) \]
\[ Y_{ZPred}(a < b) \equiv Y_{ZExpr}(a) < Y_{ZPred}(b) \]
\[ Y_{ZPred}(a \neq b) \equiv Y_{ZExpr}(a) \neq Y_{ZPred}(b) \]
\[ Y_{ZPred}(a = b) \equiv Y_{ZExpr}(a) = Y_{ZExpr}(b) \]
\[ Y_{ZPred}(\neg a) \equiv \text{not } (Y_{ZPred}(a)) \]
\[ Y_{ZPred}(a \land b) \equiv Y_{ZPred}(a) \text{and } Y_{ZPred}(b) \]
\[ Y_{ZPred}(a \lor b) \equiv Y_{ZPred}(a) \text{or } Y_{ZPred}(b) \]
\[ Y_{ZPred}(\text{True}) \equiv \text{true} \]
\[ Y_{ZPred}(\text{False}) \equiv \text{true} \]
\[ Y_{ZPred}(a \in B) \equiv \text{member } (Y_{ZExpr}(a), Y_{ZExpr}(B)) \]
\[ Y_{ZPred}(a) \equiv a \]
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* )
\[
(\Box i \bullet a_i \rightarrow A_i) \parallel ns_1 | cs | ns_2 (\Box j \bullet b_j \rightarrow B_j)
\]
\[
= (\Box i \bullet a_i \rightarrow A_i) \parallel ns_1 | cs | ns_2 ((\Box j \bullet b_j \rightarrow B_j) (c \rightarrow C))
\]

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’ )
\[
(A_1 \parallel ns_1 | cs_1 | ns_2 \parallel A_2 (c)) \setminus cs_2
\]
\[
= ((c! e \rightarrow A_1) \parallel ns_1 | cs_1 | ns_2 \parallel (c? x \rightarrow A_2 (x))) \setminus cs_2
\]

provided

- \( c \in cs_1 \)
- \( c \in cs_2 \)
- \( x \notin FV(A_2) \)

Law 3 (Sequence unit)
\[
(A) \text{Skip}; A
\]
\[
(B) A = A; \text{Skip}
\]

Law 4 (True guard)
\[
(true) \land A = A
\]
Law 5 (External choice/Sequence—distribution)

\( (\square i \bullet (g_i \& c_i \rightarrow A_i)); B = \square i \bullet (g_i \& c_i \rightarrow A_i); B \)

Law 6 (Parallelism composition/External choice—distribution*)

\( \square i \bullet (A_i [n_{s1} | cs | n_{s2}] A) = (\square i \bullet A_i) [n_{s1} | cs | n_{s2}] A \)

provided

- \(initials(A) \subseteq cs\)
- \(A\) is deterministic

Law 7 (Hiding/External choice—distribution*)

\( (A_1 \square A_2) \setminus cs = (A_1 \setminus cs) \sqcup (A_2 \setminus cs)\)

provided

- \((initials(A_1) \cup initials(A_2)) \cap cs = \emptyset\)

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

- \(cs \cap usedC(A) = \emptyset\)

Law 10 (Parallelism composition/Sequence—step*)

\( (A_1 ; A_2) [n_{s1} | cs | n_{s2}] A_3 = A_1 ; (A_2 [n_{s1} | cs | n_{s2}] A_3)\)

provided

- \(initials(A_3) \subseteq cs\)
- \(cs \cap usedC(A_1) = \emptyset\)
- \(wrtV(A_1) \cap usedV(A_3) = \emptyset\)
- \(A_3\) is divergence-free
- \(wrtV(A_1) \subseteq n_{s1}\)

Law 11 (Variable Substitution2*)

\( \text{var } x \bullet A(x) = \text{var } y \bullet A(y) \)

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*)

$$\text{var } x : T \cdot A = AA = \text{var } x : T \cdot A$$

provided $x \notin FV(A)$

Law 15 (Guard combination) 1

$$(g_1) \& ((g_2) \& A) = ((g_1 \land g_2)) \& A$$

Law 16 (Variable block/Sequence—extension*) 1

$$A_1 ; (\text{var } x : T \cdot A_2) ; A_3 = (\text{var } x : T \cdot A_1 ; A_2 ; A_3)$$

provided $x \notin FV(A_1) \cup FV(A_3)$

Law 17 (prom-var-state) 0]

\[
\begin{align*}
\text{begin} & \quad \text{begin} \\
(\text{state } S) & \quad (\text{state } S \land [x : T]) \\
L(x : T) & \quad L(\omega) \\
\bullet (\text{var } x : T \cdot MA) & \quad \bullet MA \\
\text{end} & \quad \text{end}
\end{align*}
\]

Law 18 (prom-var-state-2) 0]

\[
\begin{align*}
\text{begin} & \quad \text{begin} \\
L(x : T) & \quad (\text{state}[x : T]) \\
\bullet (\text{var } x : T \cdot MA) & \quad \bullet MA \\
\text{end} & \quad \text{end}
\end{align*}
\]

Law 19 (Guard/Parallelism composition—distribution*) 3

$$(g) \& A_1 \parallel ns_1 \mid cs \parallel ns_2 \mid A_2 = (g) \& (A_1 \parallel ns_1 \mid cs \parallel ns_2 \mid A_2)$$

provided initials($A_2$) $\subseteq cs$

Law 20 (Process Splitting 3 ArturGomes)

provided $x$ is a local variable, and therefore, can be promoted into a parameter of the process $P$

\[
\begin{align*}
\text{process } G & \equiv \text{begin } PPar \bullet \text{var } x : T \bullet P(x) \text{end} \\
& = \\
\text{process } G & \equiv \text{begin } PPar \bullet \sqcap x : T \bullet P(x) \text{end} \\
& = \\
\text{process } G & \equiv \sqcap x : T \bullet G'(x) \\
\text{process } G' & \equiv (x : T \bullet \text{begin } PPar \bullet P(x) \text{end})
\end{align*}
\]
Law 21 (Process Splitting 4 ArturGomes)

\[
\text{process } G \equiv \begin{pmatrix}
\begin{array}{c}
\text{begin} \\
\text{PPar} \\
\bullet \text{var } x : T \bullet P(x)
\end{array}
\end{pmatrix}
\]

\[
= \text{process } G \equiv x : T \bullet \begin{pmatrix}
\begin{array}{c}
\text{begin} \\
\text{PPar} \\
\bullet P(x)
\end{array}
\end{pmatrix}
\]

D.1 Lemmas from Deliverable 24.1 [132]

Lemma (K.2)

\[
\begin{pmatrix}
A \\
\emptyset \mid \text{MEM}_I \mid \{b\}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
A \\
\emptyset \mid \text{MEM}_I \mid \{b\}
\end{pmatrix}
\]

\[
\begin{array}{c}
\boxdot n : \text{NAME} \bullet \text{mget}.n!b(n) \rightarrow \text{Memory}(b) \\
\boxdot (\boxdot n : \text{NAME} \bullet \text{mset}.n?nv \rightarrow \text{Memory}(b \oplus \{n \mapsto nv\})) \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\]

\[
\setminus \text{MEM}_I
\]

D.2 Circus Denotational Semantics [129, Chapter 3]

Definition 1 (B.41)

\[
\text{begin state } [\text{decl} \mid \text{pred}] \text{PPars} \bullet A \text{ end } \equiv \text{var decl} \bullet A
\]
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 libraries. 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 \((T_1 \ast T_2 \ast T_3)\) into \(((T_1 \ast T_2) \ast T_3)\), 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 \(\Omega\) functions along with the example of the \(\Omega'\) 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 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.

```haskell
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_prime_CAction cb))
omega_prime_CAction x = omega_CAction x
```

---

128
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))" |
"omega_CAction (CSPHide a cs) = (CSPHide (omega_CAction a) cs)" |
"omega_CAction x = x" 
where
lsx = remdups.concat.map get_ZVar_st (free_var_CAction (CSPExtChoice ca cb)) |
"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
```

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 a) :|: (toUTP b)" |
| "toUTP CSPSeq a b = (toUTP a) ;; (toUTP b)" |
| "toUTP CSPIntChoice a b = (toUTP a) ∷ (toUTP b)" |
| "toUTP CSPRepSeq xs p = ( (::| xs | xs ⋆ (toUTP p)))" |
| "toUTP CSPCommAction (ChanComm c Nil) d = PrefixCSP d ⇒ (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 \textit{toUTP} to the Haskell description of that action, \((\text{CSPSeq CSPSkip CSPSkip})\), it can be refined into \textit{Skip} in the syntax of Isabelle/UTP.

\begin{verbatim}
lemma "toUTP (CSPSeq CSPSkip CSPSkip) ⊑ Skip"
apply simp
apply (metis CSP3_Skip CSP3_def Healthy_if eq_refl)
done
\end{verbatim}

The proof first transforms the left-hand side into the syntax of Isabelle, and then, using the built-in theories of \textit{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.

\subsection*{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 \textit{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 \textit{Circus} and Z schemas using ZRC. Once able to verify the correctness of our tool, we could start implementing a module of \textit{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 \(P ⊑ Q\), \(P\) is refined by \(Q\) by rewriting them using the intermediate steps proved as lemmas.

We illustrate below an example of one of the \textit{Circus} refinement laws, initially implemented in Haskell, then imported to Isabelle. First we introduce how the \textit{Refinement} type is defined in Isabelle - we had to change the names of \textit{Done} and \textit{None} to \textit{Done} and \textit{NoRef}, as \textit{None} is a reserved word in the context of Isabelle. Therefore, we use \(\forall t\) \textit{Refinement} for a refinement of either a \textit{CAction} or a \textit{CProcess}.

\begin{verbatim}
datatype (\forall t) Refinement
  = NoRef
  | Done "'t option" "'t option" "ZPred list"
\end{verbatim}

Then, we illustrate the implementation of the law sequence-unit, L. 3. It starts with a \textit{Circus} action in the shape of \((\text{CSPSeq CSPSkip act})\) or \((\text{CSPSeq act CSPSkip})\), for \textit{Skip} ; \textit{Act} and \textit{Act} ; \textit{Skip} respectively, and when applied, it results in a \textit{CAction Refinement}, with only \textit{Act}.

\begin{verbatim}
(* 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"
\end{verbatim}
And the refinement of, for example, Skip; \( \text{terminate} \rightarrow \text{Skip} \), when applying \( \text{crl_seqSkipUnit}_a \) is, therefore, \( \text{terminate} \rightarrow \text{Skip} \), as illustrated in Figure F.2.

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 \textit{Circus} in Isabelle. We use the above presented equation for applying the law \textit{sequence-unit}, as illustrated in Figure F.2, and prove its equivalence to the syntax of Isabelle/UTP. The action \( \text{PrefixCSP} \ll " \text{terminate}" \gg \text{Skip} \) is a syntactic sugar for \( \text{terminate} \rightarrow \text{Skip} \). And as expected the proof is concluded using the \textit{simp} command of Isabelle/UTP.

```
lemma "toUTP (fromDone (refined (crl_seqSkipUnit_a
  (CSPSeq CSPSkip (CSPCommAction (ChanComm '"terminate'" [])) CSPSkip)))
= PrefixCSP "'terminate'" >> Skip"
  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

\[
\text{Init} = \{ \text{Init schema def} \}
\]

\[
[\Delta \text{State} \mid \text{sec}' = 0 \land \text{min}' = 0] = \{ \text{basic conversion } bC \ [34] \}
\]

\[
\text{sec, min} : ((\text{inv} \land \exists \text{sec}', \text{min}' \bullet \text{sec}' = 0 \land \text{min}' = 0 \land \text{inv}'), (\text{inv}' \land \text{sec}' = 0 \land \text{min}' = 0)) = \{ \text{predicate calculus} \}
\]

\[
\text{sec, min} : ([\exists \text{sec}', \text{min}' \bullet \text{sec}' = 0 \land \text{min}' = 0], (\text{sec}' = 0 \land \text{min}' = 0)) = \{ \text{Assignment conversion } assC \ [34], \text{provided } inv[0, 0/min, sec] \}
\]

\[
(\text{sec, min} := 0, 0)
\]

IncSec action

\[
\text{IncSec} = \{ \text{Init schema def} \}
\]

\[
[\Delta \text{State} \mid \text{sec}' = (\text{sec} + 1)] = \{ \text{basic conversion } bC \ [34] \}
\]

\[
\text{sec, min} : ((\text{inv} \land \exists \text{sec}', \text{min}' \bullet \text{sec}' = (\text{sec} + 1) \land \text{inv}'), (\text{inv}' \land \text{sec}' = (\text{sec} + 1))) = \{ \text{L 5.4 - contract frame } [119] \}
\]

\[
\text{sec} : [(\text{inv} \land \exists \text{sec}', \text{min}' \bullet \text{sec}' = (\text{sec} + 1) \land \text{inv}'][\text{min}/\text{min}'], (\text{inv}' \land \text{sec}' = (\text{sec} + 1))[\text{min}/\text{min}']] = \{ \text{predicate calculus} \}
\]

\[
\text{sec} : ([\exists \text{sec}' \bullet \text{sec}' = (\text{sec} + 1)), (\text{sec}' = (\text{sec} + 1))] = \{ \text{Assignment conversion } assC \ [34], \text{provided } inv[(\text{sec} + 1)/\text{sec}] \}
\]

\[
(\text{sec} := \text{sec} + 1)
\]
**IncMin action**

\[ \text{IncMin} = \begin{cases} \text{Init schema def} \\ \Delta \text{State} | \text{min'} = (\text{min} + 1) \end{cases} \]

\[ \text{sec, min} : [(\text{inv} \land \exists \text{sec'}, \text{min'} \land \text{min}' = (\text{min} + 1) \land \text{inv'}) \land (\text{inv} \land \text{min}' = (\text{min} + 1))] \]

\[ \text{L 5.4 - contract frame \[119\]} \]

\[ \text{sec} : [(\exists \text{min'} \land \text{min'} = (\text{min} + 1)), (\text{min}' = (\text{min} + 1))] \]

\[ \text{Assignment conversion assC [34], provided inv[(min + 1)/min]} \]

\( \text{(min := min} + 1) \)

**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 \( \Omega \) functions as well as the refinement laws.

**G.2.1 Assignment then choice - Both sides**

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

\[ P_S.((c_1 \rightarrow x := 0) \quad \Box \quad (c_2 \rightarrow x := 1)) \]

**Proof**

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

\[ = P. \]

\[ \text{var } b : \{ y : \text{BINDING} \mid b(x) \in \text{N} \land \text{inv}(b(x)) \} \]

\[ \begin{cases} (x := 0; c_1 \rightarrow \text{Skip}) \\ (x := 1; c_2 \rightarrow \text{Skip}) \end{cases} \]

\[ \text{terminate } \rightarrow \text{Skip} \]

\[ \text{\Box} \mid \text{MEM}_I \mid \{ b \} \]

\[ \text{Memory}(b) \]

\[ \Omega_P \]

\[ \Omega_A-12 \]

\[ = P. \]
\begin{align*}
\text{var } b : \{ y : BINDING \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\bullet
&= P. \\
\text{var } b : \{ y : BINDING \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\bullet
&= P. \\
\text{var } b : \{ y : BINDING \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\bullet
&= P. \\
\text{provided:} \\
\{ \text{terminate} \} \subseteq \text{MEM}_I \\
\{ \text{mset, mget} \} \subseteq \text{MEM}_I \\
\text{mset} \in \text{MEM}_I \land \text{terminate} \in \text{MEM}_I \\
\{ \text{mset, terminate} \} \not\subseteq \{ \text{mget} \} \\
\text{provided:} \\
\{ \text{mget} \} \subseteq \text{MEM}_I \\
b \not\in \text{FV(Memory}(b)) \\
&= P.
\end{align*}
\[
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( \left( \begin{array}{c}
(\text{mset} \cdot x.0 \rightarrow \text{Skip}; \\
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \\
(\text{mset} \cdot x.1 \rightarrow \text{Skip}; \\
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\right) ; \left( \begin{array}{c}
\not\exists \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b)
\end{array} \right)
\right) \setminus \text{MEM}_I
\end{align*}
\]

\[= P.\]

\[
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( \left( \begin{array}{c}
(\text{True}) \land (\text{mset} \cdot x.0 \rightarrow \text{Skip}; \\
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \\
(\text{True}) \land (\text{mset} \cdot x.1 \rightarrow \text{Skip}; \\
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\right) ; \left( \begin{array}{c}
\not\exists \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b)
\end{array} \right)
\right) \setminus \text{MEM}_I
\end{align*}
\]

\[= P.\]

\[
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( \left( \begin{array}{c}
(\text{True}) \land (\text{mset} \cdot x.0 \rightarrow \text{Skip}; \\
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \\
(\text{True}) \land (\text{mset} \cdot x.1 \rightarrow \text{Skip}; \\
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\right) ; \left( \begin{array}{c}
\not\exists \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b)
\end{array} \right)
\right) \setminus \text{MEM}_I
\end{align*}
\]

\[= P.\]

\[
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( \left( \begin{array}{c}
(\text{mset} \cdot x.0 \rightarrow \text{Skip}; \\
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \\
(\text{mset} \cdot x.1 \rightarrow \text{Skip}; \\
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\right) ; \left( \begin{array}{c}
\not\exists \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b)
\end{array} \right)
\right) \setminus \text{MEM}_I
\end{align*}
\]

\[\text{provided:}
\begin{align*}
\text{initials(Memory}(b)) \subseteq \text{MEM}_I \\
\text{Memory}(b) \text{ is deterministic}
\end{align*}
\]

\[= P.\]

\[
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( \left( \begin{array}{c}
(\text{mset} \cdot x.0 \rightarrow \text{Skip}; \\
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \\
(\text{mset} \cdot x.1 \rightarrow \text{Skip}; \\
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\right) ; \left( \begin{array}{c}
\not\exists \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b)
\end{array} \right)
\right) \setminus \text{MEM}_I
\end{align*}
\]

\[\text{Lemma D.1}\]

\[= P.\]
\begin{align*}
\text{var} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\text{var} \ b : \text{BINDING} & \bullet \\
\begin{cases}
\begin{align*}
\text{mset}.x.0 & \rightarrow \text{Skip} ; \\
\text{c}1 & \rightarrow \text{Skip} ; \\
\text{terminate} & \rightarrow \text{Skip}
\end{align*}
\end{cases} & [\emptyset \mid \text{MEM}_I \mid \{ b \} ] \\
\begin{cases}
\begin{align*}
\text{mset}.x.1 & \rightarrow \text{Skip} ; \\
\text{c}2 & \rightarrow \text{Skip} ; \\
\text{terminate} & \rightarrow \text{Skip}
\end{align*}
\end{cases} & \text{[mset, terminate] \subseteq MEM}_I \\
\begin{cases}
\begin{align*}
\text{mset}.x?0 & \rightarrow \text{Memory}(b \oplus \{ x \mapsto 0 \})
\end{align*}
\end{cases} & \text{mem} [L.1] \\
\begin{cases}
\begin{align*}
\text{mset}.x?1 & \rightarrow \text{Memory}(b \oplus \{ x \mapsto 1 \})
\end{align*}
\end{cases} & \text{mem} [L.2+L.3]
\end{align*}
\begin{align*}
\text{provided:} & \\
\{ \text{terminate} \} & \subseteq \text{MEM}_I \\
\{ \text{mget, mset} \} & \subseteq \text{MEM}_I \\
\{ \text{mset, terminate} \} & \subseteq \text{MEM}_I \\
\{ \text{mset, terminate} \} & \not\subseteq \{ \text{mset} \} \\
& = P.
\begin{align*}
\text{var} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\begin{cases}
\begin{align*}
\text{mset}.x.0 & \rightarrow \text{Skip} ; \\
\text{c}1 & \rightarrow \text{Skip} ; \\
\text{terminate} & \rightarrow \text{Skip}
\end{align*}
\end{cases} & [\emptyset \mid \text{MEM}_I \mid \{ b \} ] \\
\begin{cases}
\begin{align*}
\text{mset}.x.1 & \rightarrow \text{Skip} ; \\
\text{c}2 & \rightarrow \text{Skip} ; \\
\text{terminate} & \rightarrow \text{Skip}
\end{align*}
\end{cases} & \text{mem} [L.2+L.3] \\
\begin{cases}
\begin{align*}
\text{mset}.x?0 & \rightarrow \text{Memory}(b \oplus \{ x \mapsto 0 \})
\end{align*}
\end{cases} & \text{mem} [L.10]
\end{align*}
\begin{align*}
\text{provided:} & \\
\{ \text{mset} \} & \subseteq \text{MEM}_I \\
x & \not\in \text{FV(\text{Memory}(b))} \\
& = P.
\begin{align*}
\text{var} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\begin{cases}
\begin{align*}
\text{mset}.x.0 & \rightarrow \text{Skip} ; \\
\text{c}1 & \rightarrow \text{Skip} ; \\
\text{terminate} & \rightarrow \text{Skip}
\end{align*}
\end{cases} & [\emptyset \mid \text{MEM}_I \mid \{ b \} ] \\
\begin{cases}
\begin{align*}
\text{mset}.x.1 & \rightarrow \text{Skip} ; \\
\text{c}2 & \rightarrow \text{Skip} ; \\
\text{terminate} & \rightarrow \text{Skip}
\end{align*}
\end{cases} & \text{mem} [L.10] \\
\begin{cases}
\begin{align*}
\text{mset}.x?0 & \rightarrow \text{Memory}(b \oplus \{ x \mapsto 0 \})
\end{align*}
\end{cases} & \text{mem} [L.10] \\
\begin{cases}
\begin{align*}
\text{mset}.x?1 & \rightarrow \text{Memory}(b \oplus \{ x \mapsto 1 \})
\end{align*}
\end{cases} & \text{mem} [L.10]
\end{align*}
\end{align*}
provided:
\begin{align*}
  \text{initials}(\text{Memory}(b)) & \subseteq \text{MEM}_t \\
  \text{MEM}_t \cap \text{usedC}(c_1 \rightarrow \text{Skip}) & = \emptyset \\
  \text{MEM}_t \cap \text{usedC}(c_2 \rightarrow \text{Skip}) & = \emptyset \\
  \text{wrtV}(c_1 \rightarrow \text{Skip}) \cap \text{usedV}(\text{Memory}(b)) & = \emptyset \\
  \text{wrtV}(c_2 \rightarrow \text{Skip}) \cap \text{usedV}(\text{Memory}(b)) & = \emptyset \\
  \text{Memory}(b) & \text{ is divergence free} \\
  \text{wrtV}(c_1 \rightarrow \text{Skip}) & \subseteq \emptyset \\
  \text{wrtV}(c_2 \rightarrow \text{Skip}) & \subseteq \emptyset \\
= P.
\end{align*}
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} & \bullet \\
& \begin{pmatrix}
(c_1 \rightarrow \text{Skip}); \\
& \begin{pmatrix}
(\text{terminate} \rightarrow \text{Skip}) \\
& \begin{pmatrix}
\emptyset \mid \text{MEM}_t \mid \{ b \} \\
\text{Memory}(b \oplus \{ x \mapsto 0 \})
\end{pmatrix} \\
\end{pmatrix} \\
\end{pmatrix} \\
\end{align*}
\begin{align*}
\text{\{L.7\}}
\end{align*}
\begin{align*}
= P.
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} & \bullet \\
& \begin{pmatrix}
(c_1 \rightarrow \text{Skip}); \\
& \begin{pmatrix}
(\text{terminate} \rightarrow \text{Skip}) \\
& \begin{pmatrix}
\emptyset \mid \text{MEM}_t \mid \{ b \} \\
\text{Memory}(b \oplus \{ x \mapsto 0 \})
\end{pmatrix} \\
\end{pmatrix} \\
\end{pmatrix} \setminus \text{MEM}_t \\
\end{align*}
\begin{align*}
\text{\{L.8+L.9\}}
\end{align*}
\begin{align*}
\text{provided:} \\
\text{MEM}_t \cap \text{usedC}(c_1 \rightarrow \text{Skip}) \cap \text{usedC}(c_2 \rightarrow \text{Skip}) & = \emptyset \\
= P.
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} & \bullet \\
& \begin{pmatrix}
(c_1 \rightarrow \text{Skip}); \\
& \begin{pmatrix}
(\text{terminate} \rightarrow \text{Skip}) \\
& \begin{pmatrix}
\emptyset \mid \text{MEM}_t \mid \{ b \} \\
\text{Memory}(b \oplus \{ x \mapsto 0 \})
\end{pmatrix} \\
\end{pmatrix} \\
\end{pmatrix} \setminus \text{MEM}_t \\
\end{align*}
\begin{align*}
\text{\{L.3\}}
\end{align*}
\begin{align*}
= P.
\end{align*}
\[
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \\
\begin{align*}
(c_1 &\rightarrow \text{Skip}) ; \\
&\qquad \left( \begin{array}{c}
\left(\text{Skip ; terminate } \rightarrow \text{Skip} \right) \\
\left( \left[ \emptyset \mid \text{MEM}_1 \mid \{b\} \right] \right) \setminus \text{MEM}_1 \\
\end{array} \right)
\end{align*}
\]

provided: \{mset\} \subseteq \text{MEM}_1 \\
x \notin \text{FV}(\text{Memory}(b)) \\
\Rightarrow P.

\[
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \\
\begin{align*}
(c_1 &\rightarrow \text{Skip}) ; \\
&\qquad \left( \begin{array}{c}
\left( mset.x.0 \rightarrow \text{Skip ; terminate } \rightarrow \text{Skip} \right) \\
\left( \left[ \emptyset \mid \text{MEM}_1 \mid \{b\} \right] \right) \setminus \text{MEM}_1 \\
\end{array} \right)
\end{align*}
\]

provided: \{terminate\} \subseteq \text{MEM}_1 \\
\{mget, mset\} \subseteq \text{MEM}_1 \\
\{mset, terminate\} \subseteq \text{MEM}_1 \\
\{mset, terminate\} \not\subseteq \{mset\} \\
\Rightarrow P.

\[
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \\
\begin{align*}
(c_1 &\rightarrow \text{Skip}) ; \\
&\qquad \left( \begin{array}{c}
\left( mset.x.0 \rightarrow \text{Skip ; terminate } \rightarrow \text{Skip} \right) \\
\left( \left[ \emptyset \mid \text{MEM}_1 \mid \{b\} \right] \right) \setminus \text{MEM}_1 \\
\end{array} \right)
\end{align*}
\]

[Lemma D.1]
var $b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}$ •
\[
\begin{aligned}
(c_1 & \rightarrow \text{Skip}) ; \\
& \left( \begin{array}{c}
(\text{mset}.x.0 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}) \\
[\emptyset | \text{MEM}_I \mid \{ b \}] \\
\text{Memory}(b)
\end{array} \right) \\
& \setminus \text{MEM}_I
\end{aligned}
\]
\[
\equiv
\begin{aligned}
(c_2 & \rightarrow \text{Skip}) ; \\
& \left( \begin{array}{c}
(\text{mset}.x.1 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}) \\
[\emptyset | \text{MEM}_I \mid \{ b \}] \\
\text{Memory}(b)
\end{array} \right) \\
& \setminus \text{MEM}_I
\end{aligned}
\]
provided:
\[
\text{MEM}_I \cap \text{used}C(c_1 \rightarrow \text{Skip}) \cap \text{used}C(c_2 \rightarrow \text{Skip}) = \emptyset
\]
\[= P.\]

var $b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}$ •
\[
\begin{aligned}
(c_1 & \rightarrow \text{Skip}) ; \\
& \left( \begin{array}{c}
(\text{mset}.x.0 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}) \\
[\emptyset | \text{MEM}_I \mid \{ b \}] \\
\text{Memory}(b)
\end{array} \right) \\
& \setminus \text{MEM}_I
\end{aligned}
\]
\[
\equiv
\begin{aligned}
(c_2 & \rightarrow \text{Skip}) ; \\
& \left( \begin{array}{c}
(\text{mset}.x.1 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}) \\
[\emptyset | \text{MEM}_I \mid \{ b \}] \\
\text{Memory}(b)
\end{array} \right) \\
& \setminus \text{MEM}_I
\end{aligned}
\]
provided:
\[
\text{initials}(A_1) \cup \text{initials}(A_2) \cap \text{MEM}_I = \emptyset
\]
\[= P.\]

var $b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}$ •
\[
\begin{aligned}
(c_1 & \rightarrow \text{Skip}) ; \\
& \left( \begin{array}{c}
(\text{mset}.x.0 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}) \\
[\emptyset | \text{MEM}_I \mid \{ b \}] \\
\text{Memory}(b)
\end{array} \right) \\
& \setminus \text{MEM}_I
\end{aligned}
\]
provided:
\[
\text{initials}((\text{Memory}(b)) \subseteq \text{MEM}_I
\]
\[
\text{MEM}_I \cap \text{used}C(c_1 \rightarrow \text{Skip}) = \emptyset
\]
\[
\text{MEM}_I \cap \text{used}C(c_2 \rightarrow \text{Skip}) = \emptyset
\]
\[
\text{wrt}V(c_1 \rightarrow \text{Skip}) \cap \text{used}V((\text{Memory}(b)) = \emptyset
\]
\[
\text{wrt}V(c_2 \rightarrow \text{Skip}) \cap \text{used}V((\text{Memory}(b)) = \emptyset
\]
\[
\text{Memory}(b) \text{ is divergence free}
\]
\[
\text{wrt}V(c_1 \rightarrow \text{Skip}) \subseteq \emptyset
\]
\[
\text{wrt}V(c_2 \rightarrow \text{Skip}) \subseteq \emptyset
\]
\[= P.
\]

var $b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}$ •
\[
\begin{aligned}
(c_1 & \rightarrow \text{Skip}; \\
& \left( \begin{array}{c}
\text{mset}.x.0 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \\
& [\emptyset | \text{MEM}_I \mid \{ b \}] \\
& \text{Memory}(b)
\end{aligned}
\]
\[
\setminus \text{MEM}_I
\]
\[
\equiv
\begin{aligned}
(c_2 & \rightarrow \text{Skip}; \\
& \left( \begin{array}{c}
\text{mset}.x.1 \rightarrow \text{Skip}; \\
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \\
& [\emptyset | \text{MEM}_I \mid \{ b \}] \\
& \text{Memory}(b)
\end{aligned}
\]
\[
\setminus \text{MEM}_I
\]
provided:

\[ \text{initials(Memory(b))} \subseteq \text{MEM}_I \]

Memory(b) is deterministic

\[ = P. \]

\[
\begin{aligned}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\left( (c_1 \rightarrow \text{Skip}; \right. \\
& mset.x.0 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \quad \Box \\
\left( (c_2 \rightarrow \text{Skip}; \right. \\
& mset.x.1 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \left\} \right) \setminus \text{MEM}_I \\
\left[ \emptyset \mid \text{MEM}_I \mid \{ b \} \right] \text{Memory(b)} \\
\end{aligned}
\]

\[ = P. \]

\[
\begin{aligned}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\left( (\text{True}) \& \quad (c_1 \rightarrow \text{Skip}; \right. \\
& mset.x.0 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \quad \Box \\
\left( (\text{True}) \& \quad (c_2 \rightarrow \text{Skip}; \right. \\
& mset.x.1 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \left\} \right) \setminus \text{MEM}_I \\
\left[ \emptyset \mid \text{MEM}_I \mid \{ b \} \right] \text{Memory(b)} \\
\end{aligned}
\]

\[ = P. \]

\[
\begin{aligned}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\left( (\text{True}) \& \quad (c_1 \rightarrow \text{Skip}; \right. \\
& mset.x.0 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) ; \\
\left( (\text{True}) \& \quad (c_2 \rightarrow \text{Skip}; \right. \\
& mset.x.1 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \left\} \right) \setminus \text{MEM}_I \\
\left[ \emptyset \mid \text{MEM}_I \mid \{ b \} \right] \text{Memory(b)} \\
\end{aligned}
\]

\[ = P. \]

\[
\begin{aligned}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
\left( c_1 \rightarrow \text{Skip}; \right. \\
& mset.x.0 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) ; \\
\left( c_2 \rightarrow \text{Skip}; \right. \\
& mset.x.1 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \left\} \right) \setminus \text{MEM}_I \\
\left[ \emptyset \mid \text{MEM}_I \mid \{ b \} \right] \text{Memory(b)} \\
\end{aligned}
\]

provided:

\[ \{ \text{mget} \} \subseteq \text{MEM}_I \]

\[ b \not\in \text{FV(Memory(b))} \]

\[ = P. \]

\[
\begin{aligned}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \wedge \text{inv}(b(x)) \} & \bullet \\
(\text{mset.x}!vx \rightarrow \; \\
\left( (c_1 \rightarrow \text{Skip}; \right. \\
& mset.x.0 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) ; \\
\left( (c_2 \rightarrow \text{Skip}; \right. \\
& mset.x.1 \rightarrow \text{Skip} \\
& \left. \text{terminate} \rightarrow \text{Skip} \right) \left\} \right) \setminus \text{MEM}_I \\
\left[ \emptyset \mid \text{MEM}_I \mid \{ b \} \right] \text{Memory(b)} \\
\end{aligned}
\]

provided:

\[ \{ \text{terminate} \} \subseteq \text{MEM}_I \]

\[ \{ \text{mset}, \text{mget} \} \subseteq \text{MEM}_I \]

\[ \text{mset} \in \text{MEM}_I \wedge \text{terminate} \in \text{MEM}_I \]

\[ \{ \text{mset}, \text{terminate} \} \not\subseteq \{ \text{mget} \} \]
\[ P. \]
\[ \begin{align*}
\text{var } b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad & \bullet \\
\left( \left( \begin{array}{c}
\text{mget.x?\text{vx} } \\
\text{c}_1 \rightarrow \text{Skip} ; \, \text{mset.x.0 } \rightarrow \text{Skip} \\
\text{c}_2 \rightarrow \text{Skip} ; \, \text{mset.x.1 } \rightarrow \text{Skip} \\
\text{terminate } \rightarrow \text{Skip}
\end{array} \right) \right) & \quad \setminus \ \text{MEM}_I
\end{align*} \]

\[ \text{[Lemma D.1]} \]

\[ P. \]
\[ \begin{align*}
\text{var } b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad & \bullet \\
\left( \left( \begin{array}{c}
\text{mget.x?\text{vx} } \\
\text{c}_1 \rightarrow \text{Skip} ; \, \text{mset.x.0 } \rightarrow \text{Skip} \\
\text{c}_2 \rightarrow \text{Skip} ; \, \text{mset.x.1 } \rightarrow \text{Skip} \\
\text{terminate } \rightarrow \text{Skip}
\end{array} \right) \right) & \quad \setminus \ \text{MEM}_I
\end{align*} \]

\[ \text{[L.3]} \]

\[ P. \]
\[ \begin{align*}
\text{var } b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad & \bullet \\
\left( \left( \begin{array}{c}
\text{mget.x?\text{vx} } \\
\text{c}_1 \rightarrow \text{mset.x.0 } \rightarrow \text{Skip} \\
\text{c}_2 \rightarrow \text{mset.x.1 } \rightarrow \text{Skip} \\
\text{terminate } \rightarrow \text{Skip}
\end{array} \right) \right) & \quad \setminus \ \text{MEM}_I
\end{align*} \]

\[ \text{[Ω}_A^{-19}] \]

\[ P. \]
\[ \begin{align*}
\text{var } b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad & \bullet \\
\left( \left( \begin{array}{c}
\text{mget.x?\text{vx} } \\
\text{c}_1 \rightarrow \Omega_A(x := 0)) \\
\text{c}_2 \rightarrow \Omega_A(x := 1)) \\
\text{terminate } \rightarrow \text{Skip}
\end{array} \right) \right) & \quad \setminus \ \text{MEM}_I
\end{align*} \]

\[ \text{[Ω}_A^{-17}] \]

\[ P. \]
\[ \begin{align*}
\text{var } b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad & \bullet \\
\left( \left( \begin{array}{c}
\text{mget.x?\text{vx} } \\
\text{c}_1 \rightarrow \Omega_A'(x := 0)) \\
\text{c}_2 \rightarrow \Omega_A'(x := 1)) \\
\text{terminate } \rightarrow \text{Skip}
\end{array} \right) \right) & \quad \setminus \ \text{MEM}_I
\end{align*} \]

\[ \text{[Ω}_A^{-2}] \]

\[ P. \]
\[ \begin{align*}
\text{var } b : \{ y : B\text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad & \bullet \\
\left( \left( \begin{array}{c}
\text{mget.x?\text{vx} } \\
\Omega_A'(c_1 \rightarrow (x := 0)) \\
\Omega_A'(c_2 \rightarrow (x := 1)) \\
\text{terminate } \rightarrow \text{Skip}
\end{array} \right) \right) & \quad \setminus \ \text{MEM}_I
\end{align*} \]

\[ \text{[Lemma D.1]} \]
= P.

\[
\begin{aligned}
\text{var } & b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \cdot \\
& \left( \\
& \begin{array}{c}
\text{mget}.x?vx \\ \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget} \rightarrow \text{terminate} \\
\text{Memory(b)}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget}.n?nv \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right)
\end{aligned}
\]

\[\text{MEM}_I\]

provided:
\{terminate\} \subseteq \text{MEM}_I
\{mget, mget\} \subseteq \text{MEM}_I
\text{mget} \in \text{MEM}_I \land \text{terminate} \in \text{MEM}_I
\{mset, terminate\} \not\subseteq \{mget\}

\[\text{L.1}\]

= P.

\[
\begin{aligned}
\text{var } & b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \cdot \\
& \left( \\
& \begin{array}{c}
\text{mget}.x?vx \\ \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget} \rightarrow \text{terminate} \\
\text{b}\{x\}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget}.n?nv \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right)
\end{aligned}
\]

\[\text{MEM}_I\]

provided:
\{mget\} \subseteq \text{MEM}_I
\text{mset} \notin \text{FV}([\text{Memory(b)}])

\[\text{L.2}\]

= P.

\[
\begin{aligned}
\text{var } & b : \{ y : \text{BINDING} \mid (\text{var}\ b) \in \mathbb{N} \land \text{inv}(b(x)) \} \cdot \\
& \left( \\
& \begin{array}{c}
\text{mget}.x?vx \\ \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget} \rightarrow \text{terminate} \\
\text{b}\{x\}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget}.n?nv \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right)
\end{aligned}
\]

\[\text{MEM}_I\]

provided:
\text{initials(Memory(b))} \subseteq \text{MEM}_I
\text{Memory(b)} \text{ is deterministic}

\[\text{L.4+L.6 twice}\]

= P.

\[
\begin{aligned}
\text{var } & b : \{ y : \text{BINDING} \mid (\text{var}\ b) \in \mathbb{N} \land \text{inv}(b(x)) \} \cdot \\
& \left( \\
& \begin{array}{c}
\text{mget}.x?vx \\ \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget} \rightarrow \text{terminate} \\
\text{b}\{x\}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget}.n?nv \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right)
\end{aligned}
\]

\[\text{MEM}_I\]

provided:
\text{initials}(c_1 \rightarrow (x := 0)) \cup \text{initials}(c_2 \rightarrow (x := 1)) \cap \text{MEM}_I = \emptyset

\[\text{L.7}\]

= P.

\[
\begin{aligned}
\text{var } & b : \{ y : \text{BINDING} \mid (\text{var}\ b) \in \mathbb{N} \land \text{inv}(b(x)) \} \cdot \\
& \left( \\
& \begin{array}{c}
\text{mget}.x?vx \\ \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget} \rightarrow \text{terminate} \\
\text{b}\{x\}
\end{array}
\right) \\
& \left( \\
& \begin{array}{c}
\text{mget}.n?nv \\
\text{terminate} \rightarrow \text{Skip}
\end{array}
\right)
\end{aligned}
\]

\[\text{MEM}_I\]

provided:
\text{initials}(c_1 \rightarrow (x := 0)) \cup \text{initials}(c_2 \rightarrow (x := 1)) \cap \text{MEM}_I = \emptyset

\[\text{Induction Hypoteses}\]
\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
(\text{vres } k : \text{BINDING} \bullet (c_1 \rightarrow (x := 0))(b))
\]
\[
(\text{vres } k : \text{BINDING} \bullet (c_2 \rightarrow (x := 1))(b))
\]

\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
(\text{vres } k : \text{BINDING} \bullet (c_1 \rightarrow (x := 0))(b))
\]
\[
(\text{vres } m : \text{BINDING} \bullet (c_2 \rightarrow (x := 1))(b))
\]

\[ p. \]
\[
\text{provided:}
\]
\[
k \notin \text{FV}(c_2 \rightarrow (x := 1)) \land m \notin \text{FV}(c_1 \rightarrow (x := 0))
\]
\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
(\text{var } k : \text{BINDING} \bullet k := b ; (c_1 \rightarrow (x := 0)) ; b := k)
\]
\[
(\text{var } m : \text{BINDING} \bullet m := b ; (c_2 \rightarrow (x := 1)) ; b := m)
\]

\[ p. \]
\[
\text{provided:}
\]
\[
k \notin \text{FV}((c_2 \rightarrow (x := 1))(b)) \land m \notin \text{FV}((c_1 \rightarrow (x := 0))(b))
\]
\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
(\text{var } k : \text{BINDING} \bullet (c_1 \rightarrow (x := 0)) ; b := b)
\]
\[
(\text{var } m : \text{BINDING} \bullet (c_2 \rightarrow (x := 1)) ; b := b)
\]

\[ p. \]
\[
\text{provided:}
\]
\[
k \notin \text{FV}((c_2 \rightarrow (x := 1))(b)) \land m \notin \text{FV}((c_1 \rightarrow (x := 0))(b))
\]
\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
(\text{var } k : \text{BINDING} \bullet c_1 \rightarrow (x := 0))
\]
\[
(\text{var } m : \text{BINDING} \bullet c_2 \rightarrow (x := 1))
\]

\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
((c_1 \rightarrow (x := 0)) \square (c_2 \rightarrow (x := 1)))
\]

\[ p. \]
\[
\text{\textbf{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \]
\[
((c_1 \rightarrow (x := 0)) \square (c_2 \rightarrow (x := 1)))
\]

\[ \text{G.2.2 Assignment then choice - Assignment in one side} \]

\[ \Omega(\text{\textit{P}}_{\text{S}}((x := 0) ; c_1 \rightarrow \text{Skip}) \square (c_2 \rightarrow \text{Skip})) \]
\[ = \]
\[ \Omega(\text{\textit{P}}_{\text{S}}((c_1 \rightarrow x := 0) \square (c_2 \rightarrow \text{Skip}))) \]

\[ \text{Proof} \]

\[ \Omega(\text{\textit{P}}_{\text{S}}((x := 0) ; c_1 \rightarrow \text{Skip}) \square (c_2 \rightarrow \text{Skip})) \]
\[ P. \]

\[ \mathbf{\text{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \]

\[
\begin{aligned}
&\left(\begin{array}{c}
(x := 0 ; \ c_1 \rightarrow \text{Skip}) \\
\square \\
(c_2 \rightarrow \text{Skip}) \\
terminate \rightarrow \text{Skip}
\end{array}\right)
\end{aligned}
\] \( \setminus \text{MEM}_I \)

\[ (\varnothing \mid \text{MEM}_I \mid \{ b \}] \]

\[ Memory(b) \]

\[ \Omega_P \]

\[ = P. \]

\[ \mathbf{\text{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \]

\[
\begin{aligned}
&\left(\begin{array}{c}
\mathbf{mget}.x ?vx \rightarrow \\
(x := 0 ; \ c_1 \rightarrow \text{Skip}) \\
\square \\
(c_2 \rightarrow \text{Skip}) \\
terminate \rightarrow \text{Skip}
\end{array}\right)
\end{aligned}
\] \( \setminus \text{MEM}_I \)

\[ (\varnothing \mid \text{MEM}_I \mid \{ b \}] \]

\[ Memory(b) \]

\[ \Omega_A - \square \]

\[ = P. \]

\[ \mathbf{\text{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \]

\[
\begin{aligned}
&\left(\begin{array}{c}
\mathbf{mget}.x ?vx \rightarrow \\
\mathbf{mset}.x.0 \rightarrow \text{Skip}; \\
\quad c_1 \rightarrow \text{Skip} \\
\square \\
(c_2 \rightarrow \text{Skip}) \\
terminate \rightarrow \text{Skip}
\end{array}\right)
\end{aligned}
\] \( \setminus \text{MEM}_I \)

\[ (\varnothing \mid \text{MEM}_I \mid \{ b \}] \]

\[ Memory(b) \]

\[ \Omega_A - := \]

\[ = P. \]

\[ \mathbf{\text{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \]

\[
\begin{aligned}
&\left(\begin{array}{c}
\mathbf{mget}.x ?vx \rightarrow \\
\mathbf{mset}.x.0 \rightarrow \text{Skip}; \\
\quad c_1 \rightarrow \text{Skip} \\
\square \\
(c_2 \rightarrow \text{Skip}) \\
terminate \rightarrow \text{Skip}
\end{array}\right)
\end{aligned}
\] \( \setminus \text{MEM}_I \)

\[ (\varnothing \mid \text{MEM}_I \mid \{ b \}] \]

\[ Memory(b) \]

\[ \text{[Lemma K.2 from Deliverable 24.1]} \]

\[ = P. \]

\[ \mathbf{\text{var}} \ b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \quad \bullet \]

\[
\begin{aligned}
&\left(\begin{array}{c}
\mathbf{mget}.x ?vx \rightarrow \\
\mathbf{mset}.x.0 \rightarrow \text{Skip}; \\
\quad c_1 \rightarrow \text{Skip} \\
\square \\
(c_2 \rightarrow \text{Skip}) \\
terminate \rightarrow \text{Skip}
\end{array}\right)
\end{aligned}
\] \( \setminus \text{MEM}_I \)

\[ (\varnothing \mid \text{MEM}_I \mid \{ b \}] \]

\[ Memory(b) \]

\[ \text{[L.1]} \]

\[ \text{provided:} \]

\[ \{ \text{terminate} \} \subseteq \text{MEM}_I \]

\[ \{ \text{mset}, \text{mget} \} \subseteq \text{MEM}_I \]

\[ mset \in \text{MEM}_I \land \text{terminate} \in \text{MEM}_I \]

\[ \{ \text{mset}, \text{terminate} \} \subseteq \{ \text{mget} \} \]

\[ \mathbf{144} \]
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \Rightarrow \\
\left( \begin{array}{c}
\text{mget.x!} \nu x \rightarrow \\
\text{mset.x.0} \rightarrow \text{Skip;}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
[\square \mid \text{MEM}_I \mid \{ b \}] \Rightarrow \\
\text{mget.x!} \nu x \rightarrow \text{Memory}(b) \end{align*}

\text{provided:} \{ \text{mget} \} \subseteq \text{MEM}_I = P.

\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \Rightarrow \\
\left( \begin{array}{c}
\text{mset.x.0} \rightarrow \text{Skip;}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
[\square \mid \text{MEM}_I \mid \{ b \}] \Rightarrow \\
\text{mset.x!} \nu x \rightarrow \text{Memory}(b)

= P.

\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \Rightarrow \\
\left( \begin{array}{c}
\text{True}\& \\
\text{mset.x.0} \rightarrow \text{Skip;}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_1 \rightarrow \text{Skip} \\
\square
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
\text{True}\& \\
c_2 \rightarrow \text{Skip} \\
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
[\square \mid \text{MEM}_I \mid \{ b \}] \Rightarrow \\
\text{mget.x!} \nu x \rightarrow \text{Memory}(b)

= P.

\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \Rightarrow \\
\left( \begin{array}{c}
\text{True}\& \\
\text{mset.x.0} \rightarrow \text{Skip;}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_1 \rightarrow \text{Skip;}
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
\text{True}\& \\
c_2 \rightarrow \text{Skip;}
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
[\square \mid \text{MEM}_I \mid \{ b \}] \Rightarrow \\
\text{mget.x!} \nu x \rightarrow \text{Memory}(b)

= P.

\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \Rightarrow \\
\left( \begin{array}{c}
\text{mset.x.0} \rightarrow \text{Skip;}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_1 \rightarrow \text{Skip;}
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
\left( \begin{array}{c}
c_2 \rightarrow \text{Skip;}
\text{terminate} \rightarrow \text{Skip}
\end{array} \right) \Rightarrow \\
[\square \mid \text{MEM}_I \mid \{ b \}] \Rightarrow \\
\text{mget.x!} \nu x \rightarrow \text{Memory}(b)

\text{provided:} \\
\text{initials}(\text{Memory}(b)) \subseteq \text{MEM}_I

\text{Memory}(b) \text{ is deterministic}

= P.
\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\ \bullet \\
(\begin{array}{l}
(\text{mset} \cdot x \cdot 0 \rightarrow \text{Skip}; \\
\quad c_1 \rightarrow \text{Skip}; \\
\quad \text{terminate} \rightarrow \text{Skip} \\
\{ \varnothing \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b) \}
\end{array})
\end{align*}

\begin{align*}
\text{MEM}_I
\end{align*}

\[\text{Lemma K.2 Deliverable 24.1}\]

\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\ \bullet \\
(\begin{array}{l}
(\text{mset} \cdot x \cdot 0 \rightarrow \text{Skip}; \\
\quad c_1 \rightarrow \text{Skip}; \\
\quad \text{terminate} \rightarrow \text{Skip} \\
\{ \varnothing \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b) \}
\end{array})
\end{align*}

\begin{align*}
\text{MEM}_I
\end{align*}

\[\text{L.1}\]

\begin{align*}
\text{provided:} \\
\{ \text{terminate} \} \subseteq \text{MEM}_I \\
\{ \text{mget, mset} \} \subseteq \text{MEM}_I \quad = P. \\
\{ \text{mset, terminate} \} \subseteq \text{MEM}_I \\
\{ \text{mset, terminate} \} \subseteq \{ \text{mset} \}
\end{align*}

\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\ \bullet \\
(\begin{array}{l}
(\text{mset} \cdot x \cdot 0 \rightarrow \text{Skip}; \\
\quad c_1 \rightarrow \text{Skip}; \\
\quad \text{terminate} \rightarrow \text{Skip} \\
\{ \varnothing \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b) \}
\end{array})
\end{align*}

\begin{align*}
\text{MEM}_I
\end{align*}

\[\text{L.2+L.3}\]

\begin{align*}
\text{provided:} \\
\{ \text{mset} \} \subseteq \text{MEM}_I \quad = P. \\
x \notin \text{FV(Memory}(b))
\end{align*}

\begin{align*}
\text{var } b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \}\ \bullet \\
(\begin{array}{l}
\quad (c_1 \rightarrow \text{Skip}; \\
\quad \text{terminate} \rightarrow \text{Skip} \\
\{ \varnothing \mid \text{MEM}_I \mid \{ b \} \mid \text{Memory}(b) \}
\end{array})
\end{align*}

\begin{align*}
\text{MEM}_I
\end{align*}
provided:

\begin{align*}
\text{initials}(\text{Memory}(b)) & \subseteq \text{MEM}_I \\
\text{MEM}_I \cap \text{usedC}(c_1 \rightarrow \text{Skip}) & = \emptyset \\
\text{MEM}_I \cap \text{usedC}(c_2 \rightarrow \text{Skip}) & = \emptyset \\
\text{wrtV}(c_1 \rightarrow \text{Skip}) \cap \text{usedV}(\text{Memory}(b)) & = \emptyset \\
\text{wrtV}(c_2 \rightarrow \text{Skip}) \cap \text{usedV}(\text{Memory}(b)) & = \emptyset \\
\text{Memory}(b) & \text{ is divergence free} \\
\text{wrtV}(c_1 \rightarrow \text{Skip}) \cap \text{usedV}(\text{Memory}(b)) & = \emptyset \\
\text{wrtV}(c_2 \rightarrow \text{Skip}) \cap \text{usedV}(\text{Memory}(b)) & = \emptyset \\
= P.
\end{align*}

\begin{align*}
\var b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( (c_1 \rightarrow \text{Skip}) ; \begin{array}{c}
\left( (\text{terminate} \rightarrow \text{Skip}) \right) \\
\left( \emptyset \mid \text{MEM}_I \mid \{ b \} \right) \\
\left( \text{Memory}(b \oplus \{ x \mapsto 0 \}) \right)
\end{array} \right) \\
\emptyset \\
\left( (c_2 \rightarrow \text{Skip}) ; \begin{array}{c}
\left( (\text{terminate} \rightarrow \text{Skip}) \right) \\
\left( \emptyset \mid \text{MEM}_I \mid \{ b \} \right) \\
\left( \text{Memory}(b) \right)
\end{array} \right)
\end{align*}

= P.

\begin{align*}
\var b : \{ y : \text{BINDING} \mid b(x) \in \mathbb{N} \land \text{inv}(b(x)) \} \bullet \\
\left( (c_1 \rightarrow \text{Skip}) ; \begin{array}{c}
\left( (\text{terminate} \rightarrow \text{Skip}) \right) \\
\left( \emptyset \mid \text{MEM}_I \mid \{ b \} \right) \\
\left( \text{Memory}(b \oplus \{ x \mapsto 0 \}) \right)
\end{array} \right) \\
\emptyset \\
\left( (c_2 \rightarrow \text{Skip}) ; \begin{array}{c}
\left( (\text{terminate} \rightarrow \text{Skip}) \right) \\
\left( \emptyset \mid \text{MEM}_I \mid \{ b \} \right) \\
\left( \text{Memory}(b) \right)
\end{array} \right)
\end{align*}

provided:

\begin{align*}
\text{MEM}_I \cap \text{usedC}(c_1 \rightarrow \text{Skip}) \cap \text{usedC}(c_2 \rightarrow \text{Skip}) & = \emptyset \\
= P.\, b : \text{BINDING} \bullet \\
\left( (c_1 \rightarrow \text{Skip}) ; \begin{array}{c}
\left( (\text{terminate} \rightarrow \text{Skip}) \right) \\
\left( \emptyset \mid \text{MEM}_I \mid \{ b \} \right) \\
\left( \text{Memory}(b \oplus \{ x \mapsto 0 \}) \right)
\end{array} \right) \\
\emptyset \\
\left( (c_2 \rightarrow \text{Skip}) ; \begin{array}{c}
\left( (\text{terminate} \rightarrow \text{Skip}) \right) \\
\left( \emptyset \mid \text{MEM}_I \mid \{ b \} \right) \\
\left( \text{Memory}(b) \right)
\end{array} \right)
\end{align*}

= P.\, b : \text{BINDING} \bullet
\( (c_1 \rightarrow \text{Skip}); \)  
\( (\text{Skip}; \ terminate \rightarrow \text{Skip}) \)  
\( (\emptyset | \text{MEM}_I | \{b\}) \)  
\( \setminus \text{MEM}_I \)  
\[ \square \]  
\( (c_2 \rightarrow \text{Skip}); \)  
\( (\text{terminate} \rightarrow \text{Skip}) \)  
\( (\emptyset | \text{MEM}_I | \{b\}) \)  
\( \setminus \text{MEM}_I \)  

provided:  
\{mset\} \subseteq \text{MEM}_I  
x \not\in \text{FV}(\text{Memory}(b))  
= P \cdot b : \text{BINDING}  
\[ \square \]

\( (c_1 \rightarrow \text{Skip}); \)  
\( (\text{mset}.x.0 \rightarrow \text{Skip}; \ terminate \rightarrow \text{Skip}) \)  
\( (\emptyset | \text{MEM}_I | \{b\}) \)  
\( \setminus \text{MEM}_I \)  
\[ \square \]  
\( (c_2 \rightarrow \text{Skip}); \)  
\( (\text{terminate} \rightarrow \text{Skip}) \)  
\( (\emptyset | \text{MEM}_I | \{b\}) \)  
\( \setminus \text{MEM}_I \)  

provided:  
\{terminate\} \subseteq \text{MEM}_I  
\{mget, \text{mset}\} \subseteq \text{MEM}_I  
\{\text{mset}, \text{terminate}\} \subseteq \text{MEM}_I  
\{\text{mset}, \text{terminate}\} \subseteq \{\text{mset}\}  
\[ \square \]  
\[ \text{Lemma K.2 from Deliverable 24.1} \]

\[ \text{L.9+L.8} \]

\[ \text{provided:} \]
\[ \text{MEM}_I \cap \text{usedC}(c_1 \rightarrow \text{Skip}) \cap \text{usedC}(c_2 \rightarrow \text{Skip}) = \emptyset \]
= \( P.b : BINDING \)  
\[
\begin{align*}
&\left( c_1 \rightarrow \text{Skip} ; \left( mset.x.0 \rightarrow \text{Skip} ; \\
&\quad \quad \text{terminate} \rightarrow \text{Skip} \right) \right) \setminus \text{MEM}_I \\
&\left( c_2 \rightarrow \text{Skip} ; \left( \text{terminate} \rightarrow \text{Skip} \right) \right) \setminus \text{MEM}_I \\
\end{align*}
\]

provided:  
\( \text{initials}(c_1 \rightarrow \text{Skip}) \cup \text{initials}(c_2 \rightarrow \text{Skip}) \cap \text{MEM}_I = \emptyset \)  
= \( P.b : BINDING \)  
\[
\begin{align*}
&\left( c_1 \rightarrow \text{Skip} ; \left( mset.x.0 \rightarrow \text{Skip} ; \\
&\quad \quad \text{terminate} \rightarrow \text{Skip} \right) \right) \setminus \text{MEM}_I \\
&\left( c_2 \rightarrow \text{Skip} ; \left( \text{terminate} \rightarrow \text{Skip} \right) \right) \setminus \text{MEM}_I \\
\end{align*}
\]

provided:  
\( \text{initials}(\text{MEM}(b)) \subseteq \text{MEM}_I \)  
\( \text{MEM}_I \cap \text{usedC}(c_1 \rightarrow \text{Skip}) = \emptyset \)  
\( \text{wrTV}(c_1 \rightarrow \text{Skip}) \cap \text{usedV}(\text{MEM}(b)) = \emptyset \)  
\( \text{MEM}(b) \) is divergence free  
\( \text{wrTV}(c_1 \rightarrow \text{Skip}) \subseteq \emptyset \)  
= \( P.b : BINDING \)  
\[
\begin{align*}
&\left( c_1 \rightarrow \text{Skip} ; \left( mset.x.0 \rightarrow \text{Skip} ; \\
&\quad \quad \text{terminate} \rightarrow \text{Skip} \right) \right) \setminus \text{MEM}_I \\
&\left( c_2 \rightarrow \text{Skip} ; \left( \text{terminate} \rightarrow \text{Skip} \right) \right) \setminus \text{MEM}_I \\
\end{align*}
\]

provided:  
\( \text{initials}(\text{MEM}(b)) \subseteq \text{MEM}_I \)  
\( \text{MEM}(b) \) is deterministic  
= \( P.b : BINDING \)  
\[
\begin{align*}
&\left( \text{True} \right) \setminus \text{MEM}_I \\
&\left( \text{True} \right) \setminus \text{MEM}_I \\
\end{align*}
\]

provided:  
\( \text{initials}(\text{MEM}(b)) \subseteq \text{MEM}_I \)  
\( \text{MEM}(b) \) is deterministic  
= \( P.b : BINDING \)  
\[
\begin{align*}
&\left( \text{True} \right) \setminus \text{MEM}_I \\
&\left( \text{True} \right) \setminus \text{MEM}_I \\
\end{align*}
\]

= \( P.b : BINDING \)
\[
\begin{align*}
\left(\left(\left(\left(\begin{array}{l}
(T) \land \left(\begin{array}{l}
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\square(T) \land \left(\begin{array}{l}
(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)ight)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

provided:

\{mget\} \subseteq \text{MEM}_I
b \notin \text{FV}(\text{Memory}(b))

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

provided:

\{\text{terminate}\} \subseteq \text{MEM}_I
\{mset, mget\} \subseteq \text{MEM}_I
mset \in \text{MEM}_I \land \text{terminate} \in \text{MEM}_I
\{mset, \text{terminate}\} \subseteq \{\text{mget}\}

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[\text{[Lemma K.2 from Deliverable 24.1]}\]

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[\text{[Lemma K.2 from Deliverable 24.1]}\]

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[\text{[Ω4 : =]}\]

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[= P.b : B\text{INDING} \bullet \]
\[
\left(\left(\left(\left(\begin{array}{l}
mget.x?vx \rightarrow \\
(\text{c}_1 \rightarrow \text{Skip}; \\
\text{mset.x.0} \rightarrow \text{Skip})
\end{array}\right)
\square(\text{c}_2 \rightarrow \text{Skip})
\end{array}\right)
\end{array}\right)
\end{align*}
\]
\[\text{terminate} \rightarrow \text{Skip} \]
\[\{\varnothing \mid \text{MEM}_I \mid \{b\}\} \text{ Memory}(b)\]  

\[= P.b : B\text{INDING} \bullet \]
\[ P.b : BINARY \bullet \]
\[
\left( \left( (c_1 \rightarrow (x := 0)) \square (c_2 \rightarrow \text{Skip}) \right); \right) \] \setminus \text{MEM}_I \]
\[
\begin{array}{c}
\left[ \emptyset \mid \text{MEM}_I \mid \{b\} \right] \text{Memory}(b)
\end{array}
\]

\[ \Omega \left( P_S.((c_1 \rightarrow x := 0) \square (c_2 \rightarrow \text{Skip})) \right) \]
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 \textit{Circus} specification. Finally, we present the \textit{CSP} code.

H.1 Original models

\textit{AChrono}

\begin{verbatim}
RANGE == 0..3
channel tick, time
channel out : {min, sec : RANGE • (min, sec)}
process AChrono ≜
begin
  state AState ≜ [sec, min : RANGE]
  AInit ≜ [AState’ | sec’ = 0; min’ = 0]
  IncSec ≜ [ΔAState | sec’ = (sec + 1) mod 60]
  IncMin ≜ [ΔAState | min’ = (min + 1) mod 60]
  Run ≜
  \begin{pmatrix}
    tick → IncSec; \\
    (if sec = 0 → IncMin \\
    \parallel sec ≠ 0 → Skip fi)
  \end{pmatrix}
  □ time → out!(min, sec) → Skip
  • (AInit ; (μX • (Run ; X)))
end
\end{verbatim}

\textit{Chrono}

\begin{verbatim}
channel inc, minsReq
channel ans : RANGE
channelset Sync == \{ inc, minsReq, ans \}
\end{verbatim}
process Chrono ≜ begin
state State ≜ [sec : RANGE] \& [min : RANGE]
SecInit ≜ [AState′ | sec’ = 0]
IncSec ≜ [ΔAState | sec’ = (sec + 1) mod 60]
RunSec ≜
  tick → IncSec;
  (if sec = 0 → inc → Skip
   \[sec ≠ 0 → Skip\])
  □time → minsReq → ans?min → out!(min, sec) → Skip
Seconds ≜ SecInit ; (μX • (RunSec ; X))
MinInit ≜ [AState′ | sec’ = 0]
IncMin ≜ [ΔAState | sec’ = (sec + 1) mod 60]
RunMin ≜ (inc → IncMin) □(minsReq → ans!min → Skip)
Minutes ≜ MinInit ; (μX • (RunMin ; X))
• ((Seconds [sec] | Sync [min] Minutes) \ Sync)
end

Distributed processes - ChronometerFull

Seconds

process Seconds ≜ begin
state StateSeconds ≜ [sec : RANGE]
SecInit ≜ [AState′ | sec’ = 0]
IncSec ≜ [ΔAState | sec’ = (sec + 1) mod 60]
RunSec ≜
  tick → IncSec;
  (if sec = 0 → inc → Skip
   \[sec ≠ 0 → Skip\])
  □time → minsReq → ans?min → out!(min, sec) → Skip
• SecInit ; (μX • (RunSec ; X))
end

Minutes

process Minutes ≜ begin
state StateMinutes ≜ [min : RANGE]
MinInit ≜ [AState′ | sec’ = 0]
IncMin ≜ [ΔAState | sec’ = (sec + 1) mod 60]
RunMin ≜
  (inc → IncMin)
  □(minsReq → ans!min → Skip)
• MinInit ; (μX • (RunMin ; X))
end

ChronometerFull

process ChronometerFull ≜ ((Seconds [Sync] Minutes) \ Sync)
H.2 After the Z Schemas Translation

AChrono

A specification:

\[
RANGE == 0\ldots3
\]

channel tick, time
channel out : \{min, sec : RANGE \bullet (min, sec)\}
process AChrono ≜
begin
state AState ≜ [sec, min : RANGE]
AInit ≜ (sec, min := 0, 0)
IncSec ≜ (sec, min := (sec + 1) mod 3, min)
IncMin ≜ (min, sec := (min + 1) mod 3, sec)
Run ≜
\[
\begin{pmatrix}
\text{tick} & \rightarrow & \text{IncSec};
\text{if } \sec = 0 & \rightarrow & \text{IncMin}
\text{\text{\text{\text{n}}}sec \neq 0 & \rightarrow & \text{Skip fi}}
\text{\text{\text{\text{\text{\text{n}}}time} & \rightarrow & \text{out!(min, sec) \rightarrow Skip}}}
\end{pmatrix}
\]
\bullet (AInit ; (\mu X \bullet (Run ; X)))
end

Chrono

channel inc, minsReq
channel ans : RANGE
channelset Sync == {inc, minsReq, ans}
process Chrono ≜ begin
state State ≜ [sec : RANGE; min : RANGE]
SecInit ≜ (sec := 0)
IncSec ≜ (sec := (sec + 1) mod 3)
RunSec ≜
\[
\begin{pmatrix}
\text{tick} & \rightarrow & \text{IncSec};
\text{if } \sec = 0 & \rightarrow & \text{inc \rightarrow Skip}
\text{\text{\text{\text{n}}}sec \neq 0 & \rightarrow & \text{Skip fi}}
\text{\text{\text{\text{\text{\text{n}}}time} & \rightarrow & \text{minsReq \rightarrow ans?min \rightarrow out!(min, sec) \rightarrow Skip}}}
\end{pmatrix}
\]
Seconds ≜ SecInit ; (\mu X \bullet (RunSec ; X))
MinInit ≜ (min := 0)
IncMin ≜ (min := (min + 1) mod 3)
RunMin ≜ (inc \rightarrow IncMin) \Box (minsReq \rightarrow ans!min \rightarrow Skip)
Minutes ≜ MinInit ; (\mu X \bullet (RunMin ; X))
\bullet ((Seconds \{\{sec\} | Sync | \{min\} | Minutes) \setminus Sync)
end
Distributed processes - *ChronometerFull*

**Seconds**

```csp
process Seconds ≜
begin
  state StateSeconds ≜ [sec : RANGE]
  SecInit ≜ (sec := 0)
  IncSec ≜ (sec := (sec + 1) mod 3)
  RunSec ≜
    \[
    \begin{align*}
    \text{tick} & \rightarrow \text{IncSec}; \\
    \text{if } \text{sec} = 0 & \rightarrow \text{inc} \rightarrow \text{Skip} \\
    \text{false} & \rightarrow \text{Skip}
    \end{align*}
    \]
    \(\Box \text{time} \rightarrow \text{minsReq} \rightarrow \text{ans?min} \rightarrow \text{out!(min, sec)} \rightarrow \text{Skip}\)
  \bullet SecInit; (\mu X \bullet (RunSec; X))
end
```

**Minutes**

```csp
process Minutes ≜
begin
  state StateMinutes ≜ [min : RANGE]
  MinInit ≜ (min := 0)
  IncMin ≜ (min := (min + 1) mod 3)
  RunMin ≜
    \[
    \begin{align*}
    \text{inc} & \rightarrow \text{IncMin} \\
    \Box \text{minsReq} & \rightarrow \text{ans?min} \rightarrow \text{Skip}
    \end{align*}
    \]
  \bullet MinInit; (\mu X \bullet (RunMin; X))
end
```

**ChronometerFull**

```csp
process ChronometerFull ≜ ((Seconds \ Sync \ Minutes) \ Sync)
```

### H.3  \(CSP_M\) code from Circus2CSP

```csp
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_RAN = {(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:RAN(over(b_RAN, n, nv)) -> 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:RAN(over(b_RAN, n, nv), ns))
[] terminate ->
( ; bd : <b_RAN> @ ; n : <y | y <- ns, member(y, dom(bd)) > @ mset.n.apply(bd, n) -> SKIP ))
MemoryMergeRAN(b_RAN, ns) =
MemoryMergeRANVar(n, b_RAN, ns)
MemoryMerge(b_RAN, ns) =
MemoryMergeRAN(b_RAN, ns)

within ( ( ( mset.sv_min. RAN.(sv_min). RAN.0 ) ->
( let X = mget.sv_min?v_sv_min:RAN(sv_min) ->
( [] inc -> mset.sv_min. (RAN.(valueRAN(v_sv_min) + 1) % 3) -> SKIP
[] minsReq -> ans.valueRAN(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:RAN(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:RAN(over(b_RAN, n, nv), ns))
[] terminate ->
( ; bd : <b_RAN> @ ; n : <y | y <- ns, member(y, dom(bd)) > @ mset.n.apply(bd, n) -> SKIP ))
MemoryMergeRAN(b_RAN, ns) =
MemoryMergeRANVar(n, b_RAN, ns)
MemoryMerge(b_RAN, ns) =
MemoryMergeRAN(b_RAN, ns)

within ( ( ( mset.sv_sec. RAN.(sv_sec). RAN.0 ) ->
( let X = mget.sv_sec?v_sv_sec:RAN(sv_sec) ->
( [] tick -> mset.sv_sec. (RAN.(valueRAN(v_sv_sec) + 1) % 3) -> SKIP
[] minsSec -> ans.valueRAN(sv_sec) -> SKIP);
X | within X )
[] terminate -> SKIP )
[] MEMI @ Memory(b_RAN) | MEMI )
ChronometerFullAOG(b) = ( ( SecondsAOG(b) | Sync | MinutesAOG(b) ) \ Sync )

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) ]
  MemoryRAN(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 -> SKIP )
  MemoryMergeRAN(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_min.(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 ->
        ansT_sv_min ->
        mset.sv_min.(RAN.t_sv_min) ->
        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)
    ( | terminate -> SKIP )
    ( | MEML |
      MemoryMerge((sv_min,v_sv_min),(sv_sec,v_sv_sec),<sv_min>) \ MEML)
    ( | Sync |
        terminate -> SKIP )
    ( | MEML |
      MemoryMerge((sv_min,v_sv_min),(sv_sec,v_sv_sec),<sv_min>) \ MEML)
    ( | Sync |
        terminate -> SKIP )
  )
  Memory(b_RAN) =
    MemoryMergeRAN(b_RAN,ns)
  MemoryMergeRAN(b_RAN,ns) =
    MemoryMergeRAN(b_RAN,ns)
  MemoryMergeRAN(b_RAN) =
    MemoryRAN(b_RAN)
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) ]
  MemoryRAN(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 -> SKIP )
  MemoryMergeRAN(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_min.(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 ->
        ansT_sv_min ->
        mset.sv_min.(RAN.t_sv_min) ->
        mset.sv_min.(RAN.t_sv_min) ->
        mset.sv_min.(RAN.t_sv_min) ->
        mset.sv_min.(RAN.t_sv_min) ->
        mset.sv_min.(RAN.t_sv_min) ->
        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)
    ( | terminate -> SKIP )
    ( | MEML |
      MemoryMerge((sv_min,v_sv_min),(sv_sec,v_sv_sec),<sv_min>) \ MEML)
    ( | Sync |
        terminate -> SKIP )
    ( | MEML |
      MemoryMerge((sv_min,v_sv_min),(sv_sec,v_sv_sec),<sv_min>) \ MEML)
    ( | Sync |
        terminate -> SKIP )
  )
  Memory(b_RAN) =
    MemoryMergeRAN(b_RAN)
  MemoryMergeRAN(b_RAN) =
    MemoryRAN(b_RAN)
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 \( \text{min} = 1 \).

\[
\begin{align*}
\text{ALARM} & := \text{ON} \mid \text{OFF} \\
\text{channel} & \text{snooze, radioOn} \\
\text{process} & \text{WakeUp } \equiv \\
\text{begin} & \\
\text{state} & \text{WState } \equiv [\text{sec}, \text{min} : \text{RANGE}; \text{buzz} : \text{ALARM}] \\
\text{WInit } & \equiv (\text{sec}, \text{min}, \text{buzz} := 0, 0, \text{OFF}) \\
\text{WIncSec } & \equiv (\text{sec}, \text{min} := (\text{sec} + 1) \mod 3, \text{min}) \\
\text{WIncMin } & \equiv (\text{min}, \text{sec} := (\text{min} + 1) \mod 3, \text{sec}) \\
\text{WRun } & \equiv \\
& \left( \begin{array}{c}
\text{tick } \rightarrow \text{WIncSec}; \\
\left(\text{sec } = 0\right) \& \text{WIncMin} \\
\left(\text{sec } \neq 0\right) \& \text{Skip} \\
\left(\text{min } = 1\right) \& \text{radioOn } \rightarrow (\text{buzz } := \text{ON}) \\
\left(\text{time } \rightarrow \text{out!(min, sec)} \rightarrow \text{Skip} \right) \\
\left(\text{snooze } \rightarrow (\text{buzz } := \text{OFF}) \right)
\end{array} \right) \cup \{\text{tick}\} \\
\bullet (\text{WInit}; (\mu X \bullet (\text{WRun } ; X)))
\end{align*}
\]

The WakeUp model may deadlock. When testing with FDR, we observed that the signal \( \text{tick} \) 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.

process WakeUpOK ≜
begin
  state WState ≜ \{sec, min \in RANGE; buzz \in ALARM\}
  WInit ≜ (sec, min, buzz := 0, 0, OFF)
  WIncSec ≜ (sec, min := (sec + 1) \mod 3, min)
  WIncMin ≜ (min, sec := (min + 1) \mod 3, sec)
  WRun ≜ \begin{cases}
    \text{tick} \to WIncSec; & (sec = 0 \land WIncMin) \\
    \text{tick} \to WIncMin; & (sec \neq 0 \land \text{Skip}) \\
    \text{tick} \to \text{WIncMin}; & \quad (\text{time} \to \text{out!}(min, sec) \to \text{Skip}) \\
    \text{tick} \to \text{WIncSec}; & \quad \left(\text{snooze} \to \text{buzz} := \text{OFF}\right)
  \end{cases}
  \bullet (WInit; (\mu X \bullet (WRun; X)))
end

H.4.1 WakeUp using Circus2CSP

--- The universe of values ---
datatype UNIVERSE = RAN.RANGE \mid ALA.ALARM
--Conversions
valueRAN(RAN.v) = v
valueALA(ALA.v) = v
typeRAN(x) = U_RAN
typeALA(x) = U_ALA
tagRAN(x) = RAN
tagALA(x) = ALA
-- subtypes of UNIVERSE for RAN
subtype U_RAN = RAN.RANGE
-- subtypes of UNIVERSE for ALA
subtype U_ALA = ALA.ALARM
-- definition of NAME for the entire spec
datatype NAME = sv_sec | sv_min | sv_buzz
-- 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) \mid v <- typeRAN(n)) \mid n <- NAME_RAN)
-- Subtype definition for ALA
b_ALA1 = \{(sv_buzz, ALA.ON)\}
subtype NAME_ALA = sv_buzz
NAMES_VALUES_ALA = seq(seq((n,v) \mid v <- typeALA(n)) \mid n <- NAME_ALA)
-- Bindings definitions for RAN
BINDINGS_RAN = \{set(b) \mid b \in set(distCartProd(NAMES_VALUES_RAN))\}
-- Bindings definitions for ALA
BINDINGS_ALA = \{set(b) \mid b \in set(distCartProd(NAMES_VALUES_ALA))\}

--- mget, mset and terminate ---
channel mget, mset : NAME.UNIVERSE
channel terminate
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 ((|{|terminate|}|)
    mset.sv_sec.(RAN.0) ->
    mset.sv_min.(RAN.0) ->
    mset.sv_buzz.(ALA.OFF) ->
    (let
        X = tick ->
        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_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_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_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) & SKIP )
                    )
                )
            )
        )
    )
) within

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) = ( MemoryALAVar(b_ALA) |{|terminate|} MemoryRAN(b_RAN) )
within ((|{|terminate|}|)
    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_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_min) != 0) & SKIP )
        )
    )
) within

-- Experiments in FDR
assert WakeUp(b_RAN1,b_ALA1) :|{deterministic [FD]}| -- Failed
4 \textbf{assert} \text{WakeUpOK}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{deterministic} \ [\text{FD}]\] \ -- \text{Passed}

6 \textbf{assert} \text{WakeUp}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{divergence free} \ [\text{FD}]\] \ -- \text{Passed}
\textbf{assert} \text{WakeUpOK}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{divergence free} \ [\text{FD}]\] \ -- \text{Passed}

8 \textbf{assert} \text{WakeUp}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{deadlock free} \ [\text{FD}]\] \ -- \text{Passed}
\textbf{assert} \text{WakeUpOK}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{deadlock free} \ [\text{FD}]\] \ -- \text{Passed}

10 \textbf{assert} \text{WakeUp}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{FD} = \text{WakeUpOK}(\text{b\_RAN1}, \text{b\_ALA1})] \ -- \text{Failed}
\textbf{assert} \text{WakeUpOK}(\text{b\_RAN1}, \text{b\_ALA1}) \ [\text{FD} = \text{WakeUp}(\text{b\_RAN1}, \text{b\_ALA1})] \ -- \text{Failed}

HWakeUp = \text{WakeUp}(\text{b\_RAN1}, \text{b\_ALA1}) \ \{\text{tick}\}

HWakeUpOK = \text{WakeUpOK}(\text{b\_RAN1}, \text{b\_ALA1}) \ \{\text{tick}\}

14 \textbf{assert} \ HWakeUp \ [\text{FD} = \text{HWakeUpOK} \ -- \text{Passed}
\textbf{assert} \ 	ext{HWakeUpOK} \ [\text{FD} = \text{HWakeUp} \ -- \text{Passed}
In memoriam

My dear mother Vilma Maria de Oliveira Silva

☆ 18/01/1954 - † 04/04/2017