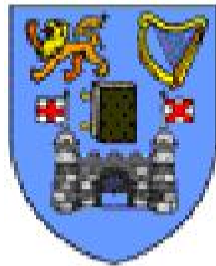
Translating from “State-Rich” to “State-Poor” Process Algebras



Department of Computer Science
Trinity College Dublin

Mirza Muhammad Arshad Beg

A Thesis Submitted to the University of Dublin, Trinity College
in Fulfillment of the Requirements for the Degree of
Doctor of Philosophy (Computer Science)

April 28, 2016

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Mirza Muhammad Arshad Beg

Summary

Following the development of formalisms based on data and behavioural aspects of the system, there are a number of attempts in which these two formalisms are mixed together to get benefit of both paradigms. *Circus* being a living specification language with continuous collaboration from both academia and industry, is a combination of Z, *CSP* and the refinement calculus. To make use of the available and industry-proven tools for a particular programming paradigm, there is a need to develop a formally verified link between one world and the other. The aim of this work is to develop a formally verified translation between the state-rich process algebra i.e. *Circus* to the state-poor process algebra i.e. *CSP*.

To achieve the research goal, the most suitable available tools had to be identified. For developing a link between targeted formal languages, the key translations required between the two languages are identified. For ensuring correctness of the translation, the key translation / refinement steps are formalised using a well-known functional language - Haskell. This formed the theoretical core of the work and supported the soundness of the link. In the end, a case study from the collection of software / hardware protocols was selected and the processes specified for the protocol were formally described using the notations available in the prototype designed in Haskell.

Dedications

Dedicated to my mother,
my wife, Amna,
and my daughters, Raafia and Rahma.

Acknowledgements

I am really grateful to my supervisor, Dr Andrew Butterfield, for his always helpful attitude during the period I conducted this research. From reviewing and fixing my early manuscripts to the underlying implementation code and the final manuscript preparation, he helped me in every aspect. His motivation drove me throughout my doctoral research. I was always impressed by the depth of his knowledge, the way of working, and appreciation of good work.

It was a real honour for me to work within such a talented group of people, especially the faculty members: Dr Matthew Hennessey, Dr Arthur Hughes and Dr Hugh Gibbons; the post-doctoral researchers of the group: Dr Vasileios Koutavas and Dr Edsko de Vries; and PhD researchers (some have already passed their PhD): Riccardo Bresciani, Pawel Gancarski, Andrea Cerone, Giovanni Bernardi, Colm Bhandal, and Carlo Spaccasassi.

Also, the acknowledgement cannot be completed without mentioning the Lero Graduate School of Software Engineering (LGSSE), which thankfully provided me the funding for my entire period of study at Trinity College Dublin.

Finally, I want to thank my parents and my wife, whom always prayed for my success. It is a good time to mention the most caring person I lost during the period of my PhD studies, my father, who always cared about me during my entire life.

Contents

1 Introduction **1**

1.1 Problem Area in General 1

1.2 Research Objective 1

1.3 Motivation and Key Inspiration 2

1.4 Notation for Language and Tool Names in the Thesis 3

1.5 Contribution of the Thesis 4

1.5.1 Semantic Justification for Turning Variables of *Circus* to Parameters in CSP_M 4

1.5.2 Developing First Version of Translator – circus2cspm 5

1.5.3 Developing Second Version of Translator in Haskell – SimpleCircus2CSPM 6

1.6 Structure of the Thesis 6

2 Background **9**

2.1 The Z notation, *CSP* and *Circus* 9

2.1.1 An Example of a program in *CSP* 9

2.1.2 Example of a Specification in *Circus* 10

2.2 Introduction to the Available Tools 12

2.3 Unifying Theory of Programming 13

2.4 The UTP Semantics of *CSP* and *Circus* 13

2.4.1 Healthiness Condition: Reactive Processes 13

2.4.2 Healthiness Conditions: *CSP* Processes 14

2.4.3 Healthiness Conditions: *Circus* Processes 14

2.4.4 Alphabets of *Circus* in UTP Semantics 14

2.4.5 Guarded Action in *Circus* : An Example 14

2.5 Dealing with Z Schemas and Assignment Command in *Circus* 15

2.5.1 Z Schemas in *Circus* 15

2.5.2 Dealing of Assignment Command 15

2.5.3 Mechanical Abstraction of CSP_Z Processes 15

2.6 *CSP* Normal Forms 16

2.6.1 A Similar Work on Normalisation of *CSP* Processes 16

2.7 Summary 17

3 State of the Art **19**

3.1 Related Work 19

3.2 Possible Case Study Area 22
3.2.1 Verification of Flash Memory Behaviour 22
3.2.2 Formal Verification of Cache Coherency Protocols 23
3.2.3 Formal Verification of Hardware Protocol – AMBA Bus Protocol 23
3.2.4 Formal Verification of Networking Protocols 23
3.3 Summary 24

4 Flash Work – An Initial Industrial Case Study 25
4.1 Introduction 25
4.2 Background 26
4.2.1 Flash Memory Operations 26
4.2.2 Host-Target Communication 26
4.2.3 The ONFi state machines 27
4.3 Related Work 27
4.4 The *CSP* Model 28
4.4.1 *CSP* Data-Entry 29
4.5 Model Analysis 29
4.5.1 Moving from ONFi 1.0 to ONFi 2.1 29
4.5.2 Running the Model in FDR 30
4.5.3 Initial Checks Performed on the Model 30
4.5.4 More Concrete Tests through Failures Refinement Checks on the Model 30
4.5.5 “Deep Hiding” along with Model Compression Techniques available in FDR 32
4.5.6 Tackling Full ONFi 2.1 Model 33
4.6 Summary 33

5 Design 35
5.1 Translation Process for a Simple *Circus* Example 35
5.2 Translation Step-by-Step 35
5.2.1 Step 1 – Scanning for the Variables in a *Circus* Process and Replacing Continuations for Skip 35
5.2.2 Step 2 – Removal of Assignment Commands and Replacing Expressions in the Parameter List 36
5.2.3 Step 3 – Analysis of Main Action and Replacing Continuations with Corresponding Calling Action 36
5.2.4 Step 4 – Inlining the Calling Actions and Propagating Parameter Changes 37
5.2.5 Increasing Complexity 37
5.3 Summary 39

6 Semantics 41
6.1 The UTP Semantics of Simple*CSP* and Simple*Circus* 41
6.1.1 Semantics of Simple*CSP* 41
6.1.2 Exploring the Semantics of Parameters 42

6.1.3 Semantics of Simple*Circus* . . . 45
6.2 The Link between Simple*Circus* and Simple*CSP* . . . 47
6.2.1 Assignment Command Handling . . . 48
6.2.2 Mathematical Proofs . . . 49
6.3 Summary . . . 55

7 Translation Theory **57**
7.1 The Template for Each Function in the Translation Process . . . 57
7.2 The Translation Process . . . 58
7.2.1 Overview . . . 58
7.2.2 Normalise Sequential Composition . . . 59
7.2.3 Rename Hidden Events . . . 59
7.2.4 Name “Next” Actions . . . 59
7.2.5 Get Variable Parameters . . . 60
7.2.6 Ensure Assignment Continutation . . . 61
7.2.7 Add Continuation Calls . . . 61
7.2.8 Propagate Assignments and Instantiate Continuations . . . 62
7.3 Summary . . . 64

8 Implementation **65**
8.1 Implementation Initial Attempt – JCircus to circus2cspm . . . 65
8.1.1 The Explanation of the Translation Flow Diagram with a Simple Example . . . 65
8.1.2 Step 1 – Specification/Environment Loading and Information Gathering . . . 66
8.1.3 Step 1.1 – Getting Channel Information . . . 67
8.1.4 Step 1.2 – Process the Z schemas to know the state variables . . . 67
8.1.5 Step 1.3 – Making each action to its parameterised version . . . 68
8.1.6 Step 2, 3 – Analysis of the Main Action . . . 68
8.1.7 Step 4 – Conversion of *CSP*-like actions to CSP_M and Alignment of Information Gathered So Far . . . 70
8.1.8 Step 5 - Final Transformation . . . 70
8.2 Haskell Implementation – SimpleCircus2CSPM . . . 72
8.2.1 Names . . . 72
8.2.2 Expressions . . . 73
8.2.3 Expression Builders . . . 73
8.2.4 Abstract Syntax of Simple*Circus* . . . 74
8.2.5 Precedence and Pretty Printing . . . 75
8.2.6 *CSP* Laws . . . 76
8.2.7 Head Normal Form Implementation in Haskell . . . 77
8.2.8 Implementation of the Step Laws . . . 78
8.2.9 Top Level Translator Function Implementation . . . 79
8.2.10 Implementation of Formalised Steps in Translation Theory . . . 83
8.3 Summary . . . 84

9 Evaluation **85**

9.1 Running Examples in SimpleCircus2CSPM . . . 85

9.1.1 Example 1 – Simple Sequential Case in the Main Action . . . 85

9.1.2 Example 2 – Adding Extra Disjoint Variables in the Main Action . . . 86

9.1.3 Example 3 – *Circus* Process without the Main Action . . . 86

9.1.4 Example 4 – Two Distinct Sequential Composition Chains of Calling Actions . . . 87

9.1.5 Example 5 – the Lift Process . . . 87

9.1.6 Example 6 – Including External Choice in the Main Action . . . 88

9.2 Case Study – A Cache Coherence Protocol Representation in Simple*Circus* Notation . . . 89

9.2.1 Background Information . . . 89

9.2.2 The Felty Example in Haskell . . . 89

9.3 Running Examples in circus2cspm Tool . . . 93

9.3.1 Example 1 – Lift Process . . . 93

9.3.2 Example 2 – Simple Sequential Chain of Calls in Initialiser . . . 94

9.3.3 Example 3 – Main Action having Internal Choice . . . 95

9.3.4 Example 4 – Varied Order of Action Calls in Example 3 . . . 96

9.3.5 Example 5 – Multiple Assignments in an Action . . . 97

9.3.6 Example 6 – Call of Actions having Sequential Composition and External Choice . . . 97

9.3.7 Example 7 – Call of Actions having External Choice and Sequential Composition . . . 98

9.3.8 Example 8 – Including Output Prefixing Action . . . 99

9.4 Summary . . . 100

10 Conclusions **101**

10.1 Extending a *CSP* Model of Flash Device Behaviour . . . 101

10.2 Development of a Prototype to Translate from *Circus* to CSP_M using Java . . . 102

10.3 Mathematical Proofs of Semantic Justification for the proposed Translation using the Unifying Theories of Programming (UTP) Semantics . . . 102

10.4 Using Haskell for Development of the Prototype for *Circus* to CSP_M Translation . . . 103

10.5 Summary . . . 104

A Introducing Karnaugh Maps for Graphical Proofs **111**

A.1 Graphical Approach . . . 111

A.1.1 Standard Reactive Diagram . . . 111

A.1.2 Logical Operations . . . 111

A.1.3 Conditionals as Projections . . . 114

A.1.4 Examples . . . 115

A.1.5 Lemma 1 – Graphical Approach . . . 117

A.1.6 Lemma 2 – Graphical Approach . . . 118

B Proofs for the Link between Simple*Circus* and Simple*CSP* **119**

B.1 A Graphical Proof Attempt . . . 119

B.2 Other Proofs . . . 123

B.3 Lemmas 130

C Haskell Implementation 131

C.1 Working with Simple Circus in Haskell 131

C.1.1 Names 131

C.1.2 Expressions 131

C.1.3 Abstract Syntax 132

C.1.4 Simplifying Constructors 133

C.2 Standard Circus Names 134

C.2.1 Names 134

C.2.2 Expression Builders 135

C.2.3 Precedence 136

C.3 Standard Circus Printing 138

C.3.1 Pretty-Printing Expressions 138

C.3.2 Pretty-Printing Processes 139

C.4 Simple Circus Translation 143

C.4.1 Name Management 143

C.5 Simple Circus Laws 145

C.5.1 Laws 145

C.5.2 Step Laws 153

C.6 Simple Circus Translation 157

C.6.1 Normal Form 157

C.6.2 Acquiring Head Normal Form 157

C.6.3 Translation 158

C.7 Simple Circus Examples 165

C.7.1 Examples 165

C.8 "Standard" Simple Circus 168

C.9 Implementation of Formalised Steps in Translation Theory 169

D Cache Coherence Protocol – Processes Specified using Simple*Circus* 179

E How to Run Examples in SimpleCircus2CSPM and circus2cspm 191

E.1 Haskell Implementation of Translator 191

E.1.1 Steps to Follow 191

E.2 Java Implementation of Translator – circus2cspm 191

E.2.1 Steps to Follow 192

Chapter 1

Introduction

This chapter describes the overall picture of the research objectives, area of contribution and technical approach adopted for our research. It also presents the layout of the whole thesis.

1.1 Problem Area in General

Software system use and its application is increasing in our daily life. For example, the number of internet users stands at 39% of total world population in 2013 [URLd]. This was standing at 16% in 2005 [URLd]. Consequently, the importance of correctness and reliability of the software is increasing [Wil09]. Errors in software design have led to major accidents in the past. For example, [URLa] lists examples of such failures. One of these examples is the Mars Climate Orbiter mission failure in 1998 [URLe]. The fault was due to an inconsistency in the units used. The designer used imperial units while NASA [URLg] was supposed to work with metric units. It resulted in a crash of the orbiter while trying to stabilise in the orbit around Mars at too low an altitude. In 2007, an error-prone piece of software running on a network card maliciously broadcasted the data on the United States Customs and Border Protection network [URLb]. It resulted in a halt of the entire system. For eight hours, nobody could leave Los Angeles airport. It resulted over 17,000 planes being grounded for the duration of the fault.

Other examples of critical software systems are software dealing with financial institutions, or software running for the medical industry and applications etc. Serious software failures and accidents in the past decades, for example in [URLa], have led software designers to realise the importance of formal methods in designing critical software systems [HM05]. By adopting formal techniques for designing software with proven correctness, the overall quality and reliability of that software can be increased.

'Formal Methods' can be defined as a collection of languages, techniques and tools based on mathematics for specifying and verifying systems [URLc]. More precisely, 'Formal Methods' is about 'formalising' a system on the basis of a set of tools and notations having a formal semantics. These tools are used to clearly specify the requirements of a system, allowing the proof of properties of that specification and to prove the correctness of an implementation with respect to that specification.

1.2 Research Objective

There has been a recent trend in the field of formal methods to link different formal methods tools e.g. model-checkers, theorem provers and model simulations, in order to make the formal verification process of a system automatic or semi-automatic, for example, the work presented in [FO09, FSO08, MAF08]. In order to utilise existing available industry proven formal methods tools, we have the option of developing automatic code

generators. The function of these automatic code generators is to translate the specifications written in one notation to the other.

The aim of this work is to design and implement a translation strategy between a "state-rich" process algebra i.e. *Circus* [WC02] to a "state-poor" process algebra i.e. machine readable CSP or CSP_M[1] [Hoa04, Ros98] and later to generalise it to have a translation strategy from an imperative language to a functional one. The need for developing links between these two languages is due to the fact that CSP_M has an industry-proven model checker [For05] while *Circus* has an under development model checker. This objective is achieved through a number of stages which are listed below:

1. To develop a formally verified link between a *Circus* based tool [MFMU05] to a CSP_M based tool [For05]. Tools in the link are to support the verification of software/hardware implementations which are derived from *Circus* specifications. This involved:
   (a) Designing and implementing a translation strategy between the two target languages.
   (b) Formalising the translation strategy to ensure the key translations' proof of correctness.
   (c) Linking the target languages by relating the semantics of the target languages. Both *Circus* and CSP_M have well-formed semantics within the Unifying Theories of Programming (UTP) framework [OCW09, CW04].
2. All the above steps ended up in a *prototype tool* for automating the translations between the two languages.
3. In order to prove the utility of the developed prototype, a case study was carried out for a selected protocol [FS96] specification by using the notation suggested in the tool.

1.3 Motivation and Key Inspiration

The key motivation for developing the link was to contribute to the 'Grand Challenge in Computing' (GC6) project. The GC6 project [HM05] was expected to: 1. deliver a comprehensive and verified theory of programming; 2. give a prototype for a comprehensive and integrated suite of programming tools; and 3. deliver a repository of verified software. So, the development of the link will be a contribution to the prototype collection of integrated suite of programming tools.

Circus is a state-rich specification language with an active research focus from the researchers as the *Circus* website [URLf] lists around 80 journal and conference publications since 2001. These publications deal with the underlying theory of the language as well as its practical use in industrial applications.

One of the key inspiration for developing this formally verified translation strategy for *Circus* based modelling is from the work [FO09]. The goal of that work was to integrate both programming and logical tools to verify formally specified operations. The authors of [FO09] verified a File System model through the combination of different formal specification languages and made their tools inter-operate together. This process was accomplished using Alloy (design and model checking), VDM++(Prototyping and Testing) and HOL (proof of correctness) as depicted in figure 1 of [FO09], shown here in figure 1.1. First of all, a highly abstract model of the architectural design of the target system was designed either by using point free notation (PF-notation) or Alloy. In spite of having no ability to prove properties, Alloy was very handy in getting counter-examples. These counter-examples were used to spot where and why the properties were failing. After validating the design in Alloy, the model was translated to VDM++ where more details were added in the model. In the VDM++ stage, validation of all functional requirements became possible. The proof of correctness of the properties of the model was perfomed with HOL. The unproved goals from both VDM++ and HOL were proved in

[1]The names *Circus* and CSP_M are formatted typographically as per the style adopted by their originators.

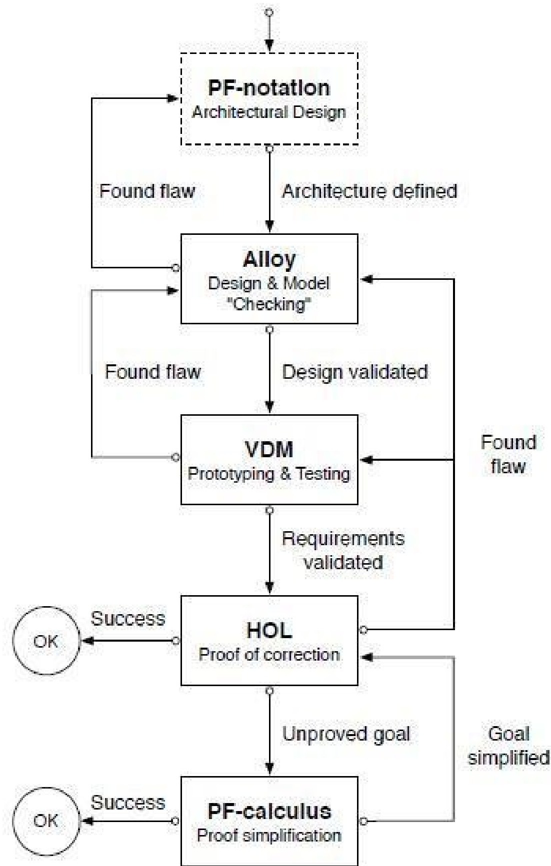


Figure 1.1: Figure 1 of [FO09]: Tool-chain Operation

the end using pen-and-paper proofs through point free (PF) calculation. This step was performed to simplify proof-obligations.

The work [HH05] deals with the linking theories of Calculus of Communicating Systems (CCS) [Mil80] and *CSP*. Here the underlying theories were linked through a Galois connection. The complete transition system of CCS was projected on the healthiness conditions of *CSP*. The strategy applied to the weak, strong and barbed simulations of CCS, along with trace, failures, and failures/divergence refinement semantics of CSP. The challenge of linking other theories of concurrency through Galois connection was also suggested in the paper [HH05]. Chapter 4 of [HH98] provides a general approach for linking theories of different languages using UTP framework.

1.4 Notation for Language and Tool Names in the Thesis

In order to maintain consistency, the following typographical style will be used for the language and tool names, referred in the text.

Notation	Explanation
circus2cspm	The prototype developed using Java for translation from *Circus* to CSP_M
SimpleCircus2CSPM	The prototype developed using Haskell for translation from *Circus* to CSP_M
Circus	Complete *Circus* language
CSP and CSP_M	Complete *CSP* and machine readable CSP language
Simple*CSP*	The subset of CSP_M selected for the mathematical proofs
Simple*Circus*	The subset of *Circus* selected for the mathematical proofs

1.5 Contribution of the Thesis

Here, we list the road path for our research and the outcomes. As our work comprises of number of stages, so we explain them here one by one.

1.5.1 Semantic Justification for Turning Variables of *Circus* to Parameters in CSP_M

A thorough explanation of the target languages is in the following chapter. Here, we show the selected subset of the abstract syntax tree of the target languages in figure 1.2. The problem area chosen is to provide the semantic justification for converting an assignment command in the *Circus* world to a parametric process in the CSP_M world, shown in figure 1.3. In case of the *Circus* process, x is assigned a value in the specification. While doing the same for CSP_M, the value is passed as a parameter. This problem is dealt in Chapter 6 of the thesis.

$$
\begin{array}{rcl}
P \in Process & ::= & A \; \mathbf{where} \; AD_1; \ldots; AD_n \\
AD \in ActDef & ::= & N \mathrel{\widehat=} A \mid N(X_1, \ldots, X_n) \mathrel{\widehat=} A \\
A \in Action & ::= & Stop \mid Skip \mid a \rightarrow A \mid e\&A \mid A \mathbin{;} A \mid A \sqcap A \mid A \mathbin{\Box} A \\
 & & \mid A \parallel A \qquad \text{— } \mathit{CSP} \text{ only} \\
 & & \mid x := e \mid A \;{}_V\!\!\parallel_V A \qquad \text{— } \mathit{Circus} \text{ only} \\
 & & \mid N \mid N(e_1, \ldots, e_n) \\
N \in Name & ::= & \text{names} \\
a, b.e \in Event & & \text{events} \\
e \in Expr & & \text{expressions} \\
x \in Var & & \text{variables} \\
V \in VarSet & & \text{variable-sets}
\end{array}
$$

Figure 1.2: *Circus*/CSP_M Syntax

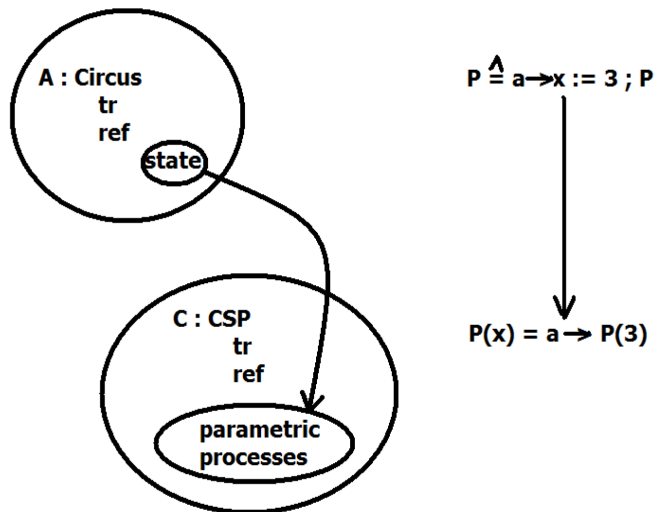


Figure 1.3: An assignment command in the *Circus* world converted to a parametric process in the CSP_M world

1.5.2 Developing First Version of Translator – circus2cspm

By utilising the CZT [2] Java classes, which contains the type checker and parser for *Circus*, the requirement of inputting *Circus* specifications was achieved. The *Circus* parser was a contribution from the work [WCF05]. In previous work [Oli05], a transformation strategy for transforming from a concrete *Circus* specification to a Java program had been proposed. It consisted of translation rules, that applied to each *Circus* construct in a concrete specification, resulting in a Java program that implements the *Circus* program. The resulting Java program uses the JCSP library, a Java implementation of the CSP_M model for concurrency and communication. The work [dF05] provided an implementation of the translation strategy. The implementation result is a tool called JCircus. This tool generates a Java implementation of the concrete *Circus* specification through a simple GUI.

The work presented in [dF05] was selected as the starting point for translating from *Circus* to CSP_M as this provided us the basis for parsing *Circus* specifications. The next step was to modify the translation rules written for conversion of Java, to work with conversion to CSP_M. In the latest version of the freely available CZT tool sources, we did not find Java sources for translation from *Circus* to CSP_M, which was an indication that it is a novel work for automatic translation of *Circus* into CSP_M. Figure 1.4 depicts the difference between the previous works available and our contribution. It is made clear in the figure that CZT classes and *Circus* parser is the contribution of Leo Freitas et. al. JCircus tool is the contribution of Angela Freitas et. al. By updating the source files of JCircus, the prototype circus2cspm is our contribution. The UTP linkage between *Circus* and CSP_M script is shown here to depict that mathematical proofs are done using UTP semantics to justify the translation between the two languages.

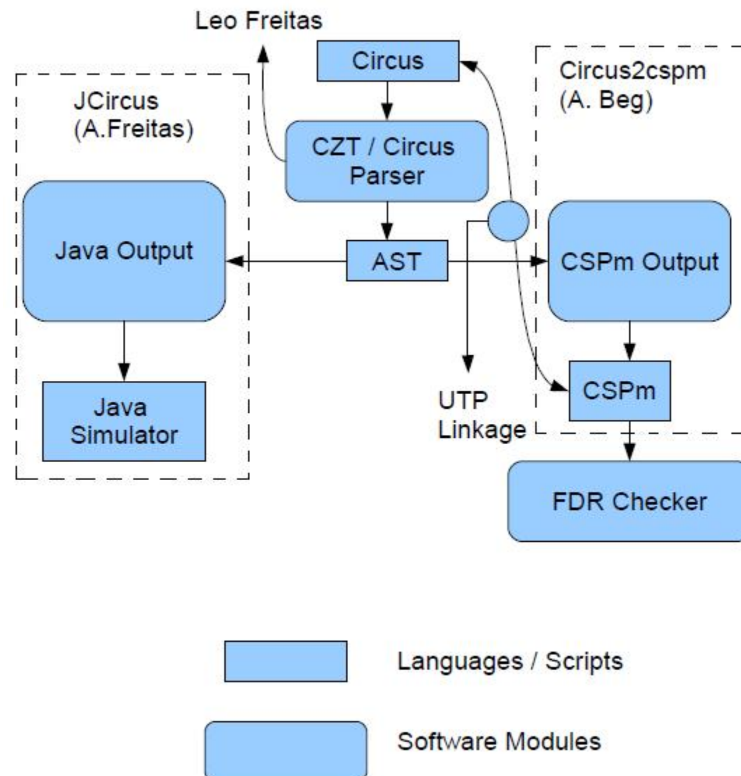


Figure 1.4: Translation from *Circus* to CSP_M

To develop the circus2cspm tool, the front end of the tool is kept the same while the back-end processing of the abstract syntax tree is updated to get the required translation. The development of this tool was done in Java using Eclipse IDE. The base design of the circus2cspm tool is based on the classes written for JCir-

[2] http://czt.sourceforge.net/manual.html

cus tool. The main modification is done in TranslatorVisitor class which visits each construct of the *Circus* language and outputs the required text. Futhermore, the Translator2Java class is renamed to Translator2CSP and modified accordingly.

The interface of circus2cspm is quite simple. It takes input from a LaTeX file. Then the user specifies the output file directory where the output project directory structure is to be created. If correctly parsed and type checked, the CSP_M output is generated. The datatype declaration is in the output file called DataType.csp where DataType is replaced with the name of actual data type given in *Circus* specification. Channel declarations are listed in ChannelDecl.csp, while the finalProcDecl.csp is the file containing the final translated version of the process definition.

1.5.3 Developing Second Version of Translator in Haskell – SimpleCircus2CSPM

Cases of some simple action compositions were tested through circus2cspm. But if this structure grows and acquires a complex form, it would be a complex task to implement the translation. Pattern matching in the compositions of complex combinations of actions in the main action is another challenge to implement in the source code of circus2cspm. This pattern matching is a pre-requisite while dealing with the implementation of all *CSP* laws in the main action. So, later on, in the implementation stage, we used Haskell – a well-known functional programming language – for modelling these translations and this is maintained in our online repository [Rep].

In the Haskell implementation for the translation, a number of examples are provided to show the use of the implementation of the translation strategy between the two worlds. We covered a number of constructs in our prototype such as prefix actions, sequential composition, internal choice, external choice, hiding action, interface parallel, alphabetised parallel, conditional, assignment command, calling an action as well as their generalised forms. A significant number (forty four) of *CSP* algebraic laws are formalised in the implementation. A normalisation process is also implemented in the prototype. A simple implementation of pretty printing is also provided which can be extended to provide a more sophisticated one.

1.6 Structure of the Thesis

Chapter 2 of the thesis provides background information for the target languages i.e. *Circus* and CSP_M. Later, it explains the Unifying Theory of Programming (UTP) [HH98]. The explanation of UTP is necessary as later the link between the two languages is proven feasible using the UTP semantics of the languages.

Chapter 3 provides state of the art section of the thesis. This chapter describes the similar works available in the literature. Our approach for the translation between *Circus* and CSP_M is by turning variables into parameters while the similar work described in the chapter turns variables into processes with set and get channels, or clocked hardware.

Chapter 4 provides the details of an industrial case study. This chapter is a published work in an ACM conference proceedings. The case study conducted is about modelling flash devices behaviour using CSP_M. The updated flash device behaviour model is model-checked using the Failures-Divergence Refinement (FDR) toolkit.

Chapter 5 explains the translation process adopted in our first attempt of developing a prototype tool using Java. It takes a simple example of the *Circus* specification and explains each stage of the translation involved. This chapter also gives some implementation details about the data structures and record keeping process for the transformation of the specification from *Circus* to CSP_M.

Chapter 6 explains the semantics of *Circus* and CSP_M in UTP framework and then gives description of our work of linking the underlying theories.

Chapter 7 provides the formalisation of the translation process between the target languages. Here, the steps

involved in the translation are formalised and presented in mathematical style.

Chapter 8 explains the implementation of the translation strategy using Java and then in Haskell. Implementation in Haskell provides the basic constructs of the languages, *CSP* laws available in [Ros98], normalisation process available in Chapter 11 of [Ros98], prototyping of the step laws involved, pretty-printing of the languages and some working examples.

Chapter 9 evaluates the translated examples using the SimpleCircus2CSPM tool and gives comments on the results achieved so far. It also describes the case study of a cache coherence protocol specified in the notation of Simple*Circus* in order to explain the usefulness of Simple*Circus* for specifying such protocols.

Chapter 10 concludes the thesis by listing the contributions presented in this work.

Appendix contents are divided into sections given below.

Section A gives an introduction to the graphical way of proof using Karnaugh Maps technique. This technique was proposed and devised by Dr Andrew Butterfield and is used in the proof of theorem B.1.1. The need for devising this technique appeared when the usual way of proof by expansion of definitions did not work for theorem B.1.1. **Section B** includes remaining mathematical proofs done for establishing the link between *Circus* and CSP_M using the UTP framework. It also includes the proposed lemmas for the proof of theorem 6.2.5. The proofs of these lemmas have been gone through but they are of significant length. So, currently they are not included in the body of the thesis. **Section C** includes the Haskell code of the implementation of the translator SimpleCircus2CSPM. **Section D** contains a case study, which includes the specification of processes for a cache coherency protocol using Simple*Circus* notation. **Section E** includes the instructions how to run and see output of the examples translated through the tools circus2cspm and SimpleCircus2CSPM.

The following chapter gives us the background information about the target languages and their semantics.

Chapter 2

Background

This chapter provides background information for the target languages, *Circus* and CSP_M. Later, it explains the unifying theory of programming (UTP) [HH98]. The explanation of UTP is necessary as later the link between the two languages is proven feasible using the UTP semantics of the languages. This chapter also explains work of others described in the literature which is relevant to this work.

2.1 The Z notation, *CSP* and *Circus*

The Z notation [WD96] has its basis in the set theory and mathematical logic. Standard set operators, set comprehensions, Cartesian products, and power sets are used in the set theory. First-order predicate calculus is used as the mathematical logic. In Z, the mathematics can be structured by the use of *schemas*. Collections of mathematical objects and their properties are described using these schemas. A unique type is given to each mathematical object in the language. So, the use of types gives the functionality of checking the type of each object in the specification. The use of refinement calculus [Cav97] is another characteristic feature of the Z language. A model is defined using simple mathematical data types to describe the desired behaviour. Then this description is refined by construction of another model. This refined model gives a description closer to the implementation of the system. The other notations similar to Z are the B-method and Object-Z (a Z extension with an object oriented approach).

Communicating Sequential Processes (*CSP*) [Hoa04] is a process algebra where the systems are represented as processes. In the *CSP* world, the system is specified as the order in which these processes are to be performed. *CSP* allows concurrency and provides a way in which these processes can interact. This is achieved through channels. Messages can be exchanged betweeen these channels from one process to the another one. However, data requirements are not handled very well in process algebras. In *CSP*, parameters can be defined for a limited data modelling. The Calculus of Communicating Systems (CCS) [Mil80] is another example of a process algebra.

2.1.1 An Example of a program in *CSP*

In this example, the door of a lift can be opened or closed, as exemplified by the type `DoorState` and there are four possible events in which the lift may engage i.e. `up, down, open` and `close`.

```
datatype DoorState = opened | closed
channel up
channel down
channel open
channel close
```

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

There are a number of efforts in which the process algebras and state-rich formal languages are mixed together to gain benefits from both paradigms. This is necessary for the systems that have both data and behavioural requirements. Some of these examples are CSP∣B [ST05], Z with *CSP* [MS01], Z with CCS [GS97] and many more. *Circus* is a specification language that combines Z, *CSP* and refinement calculus constructs. The main difference between *Circus* and other ones is that the languages, i.e. Z and *CSP*, are mixed freely in a specification. A *Circus* program consists of a sequence of paragraphs. These paragraphs can be a: Z paragraph, channel or channel set definition, or a process declaration. A process paragraph may contain a Z paragraph, an action definiton or a name set definition. The process paragraph is started and ended with the keywords **begin** and **end**.

2.1.2 Example of a Specification in *Circus*

$DoorState ::= opened \mid closed$

channel $up, down, open, close$
process $Lift \mathrel{\widehat=}$ **begin**
state

$$\begin{array}{|l}
\textit{LiftState} \\
\hline
floor : \mathbb{N} \\
doorState : DoorState \\
\hline
\end{array}$$

$InitLift \mathrel{\widehat=} (floor := 0;\ doorState := closed)$

$Lift \mathrel{\widehat=} ((floor < 5 \land doorState = closed)\ \&\ up \rightarrow floor := floor + 1)$
$\quad \Box\ ((floor > 0 \land doorState = closed)\ \&\ down \rightarrow floor := floor - 1)$
$\quad \Box\ (doorState = closed\ \&\ open \rightarrow doorState := opened)$
$\quad \Box\ (doorState = opened\ \&\ close \rightarrow doorState := closed)$

$\bullet\ InitLift;\ \mu X \bullet (Lift;\ X)$

end

If the examples of *CSP* and *Circus* are analysed critically, there are apparent differences between the two, among which a few are discussed here. In the lift example of *CSP*, the process `INITIAL_LIFT` is initiated with the call to process `LIFT` by passing parameters to it with some initial values, e.g. `LIFT(0, closed)`. On the other hand, in the *Circus* example, the state variables defined by the Z schema *LiftState* are initialized with assignment statements in the process *InitLift*. Furthermore, the same difference appears in the guarded commands of actions in a *Circus* specification, where after making a decision about a state of the variable e.g. $floor < 5$, the state variables changes its state being assigned an expression, e.g. $floor + 1$. On the other hand,

$$
\begin{array}{rcl}
\mathsf{CircusProgram} & ::= & \mathsf{CircusPara}^{*} \\
\mathsf{CircusPara} & ::= & \textbf{channel}\ \mathsf{CDecl} \\
 & | & \mathsf{ProcDecl} \\
\mathsf{CDecl} & ::= & \mathsf{N}^{+} \mid \mathsf{N}^{+} : \mathsf{Expr} \\
\mathsf{ProcDecl} & ::= & \textbf{process}\ \mathsf{N} = \mathsf{ProcDef} \\
\mathsf{ProcDef} & ::= & \mathsf{CircProc} \\
\mathsf{CircProc} & ::= & \textbf{begin}\ \mathsf{PPara}^{*}\ \textbf{state}\ \mathsf{Para}\,\mathsf{PPara}^{*} \bullet \mathsf{Action}\ \textbf{end} \\
 & | & \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred}\ \&\ \mathsf{Action} \mid \mathsf{Action} \backslash \mathsf{CSExpr} \\
 & | & \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \,\Box\, \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
 & | & \mathsf{Proc} \,|[\mathsf{CSExpr} \| \mathsf{CSExpr}]|\, \mathsf{Proc} \\
 & | & \mathsf{Proc} \,|[\mathsf{CSExpr}]|\, \mathsf{Proc} \\
 & | & \mathsf{Proc} \,|||\, \mathsf{Proc} \\
 & | & \mathsf{Decl} \bullet \mathsf{Proc} \left(\mathsf{Expr}^{+}\right) \\
 & | & (\mu\, \mathsf{N} \bullet \mathsf{Decl} \bullet \mathsf{Proc}) \left(\mathsf{Expr}^{+}\right) \\
 & | & \Box\, \mathsf{Decl} \bullet \mathsf{Proc} \\
 & | & \sqcap\, \mathsf{Decl} \bullet \mathsf{Proc} \\
 & | & |||\, \mathsf{Decl} \bullet \mathsf{Proc} \\
\mathsf{CSExpr} & ::= & \{|\ \mathsf{N}^{*}\ |\} \mid \mathsf{N} \mid \mathsf{CSExpr} \cup \mathsf{CSExpr} \\
 & | & \mathsf{CSExpr} \cap \mathsf{CSExpr} \mid \mathsf{CSExpr} \backslash \mathsf{CSExpr}
\end{array}
$$

Figure 2.1: BNF Syntax of Circus Processes

in the *CSP* world, the same operation is implemented by passing parameters e.g. `LIFT(floor + 1, doorState)`.

The BNF Syntax for *Circus* processes and actions are shown in the figures 2.1 and 2.2. These syntax trees are simplified versions. The complete version can be seen in [OCW09][1]. In the BNF syntax of *Circus* processes, $\mathsf{CircusPara}^{*}$ and PPara^{*} denote a possible empty list of elements of the syntactic category CircusPara of *Circus* paragraphs and PPara of process paragraphs. The complete BNF representation of *Circus* syntax also includes the categories SchemaExp, Exp, Pred and Decl to show the schema expressions of Z, expression lists, predicates and declarations.

N^{+} is used here to indicate a non-empty list of Z identifiers N. In the CircProc definition, **state** defines the state variables of the specification. The process definition is enclosed in the keywords **begin** and **end**. The *Circus* processes can be: a prefix action i.e. where a communication is performed followed by an action depicted as $\mathsf{Comm} \rightarrow \mathsf{Action}$; A guarded action where a predicate is checked for taking an action depicted as $\mathsf{b}\ \&\ \mathsf{Action}$; a hiding action $\mathsf{Action} \setminus \mathsf{CSExpr}$ in which members of channel sets included in CSExpr are kept hidden while performing an action; a sequential composition $\mathsf{Proc} ; \mathsf{Proc}$ where a process is followed by another process; an external choice between two processes $\mathsf{Proc} \,\Box\, \mathsf{Proc}$; an internal choice between two processes $\mathsf{Proc} \sqcap \mathsf{Proc}$; the processes are put in parallel with each other by $\mathsf{Proc} \,|[\mathsf{CSExpr} \| \mathsf{CSExpr}]|\, \mathsf{Proc}$; and the interleaving of the processes where none of the members of channel sets are common to both processes, $\mathsf{Proc} \,|||\, \mathsf{Proc}$. The BNF syntax of *Circus* actions also includes: the basic constructs of *CSP* i.e. *Skip* (Do nothing), *Stop* (termination) and *Chaos* (depicting chaotic behaviour of the program); the assignment command ($\mathsf{N} := \mathsf{e}$) where a variable is assigned an expression; the conditional between two actions i.e. if a boolean b is true, perform action on the left of conditional otherwise perform action on the right side of the conditional ($\mathsf{Action} \lhd \mathsf{b} \rhd \mathsf{Action}$); and $\mu\ \mathsf{N} \bullet \mathsf{Action}$ depicts a recursive call to an action.

[1] The typographical style of BNF syntax here is kept same as in the original cited work.

$$
\begin{array}{rcl}
\mathsf{CircusPara} & ::= & \mathsf{N} \mathrel{\widehat=} \mathsf{ActionDef} \\
\mathsf{ActionDef} & ::= & \mathsf{Action} \\
\mathsf{Action} & ::= & \mathit{Skip} \\
 & | & \mathit{Stop} \\
 & | & \mathit{Chaos} \\
 & | & \mathsf{Comm} \rightarrow \mathsf{Action} \\
 & | & \mathsf{b} \,\&\, \mathsf{Action} \\
 & | & \mathsf{Action} \sqcap \mathsf{Action} \\
 & | & \mathsf{Action} \,\Box\, \mathsf{Action} \\
 & | & \mathsf{Action};\ \mathsf{Action} \\
 & | & \mathsf{Action} \,[\![s_1 \mid \{\!| CS |\!\} \mid s_2]\!]\, \mathsf{Action} \\
 & | & \mathsf{Action} \setminus CS \\
 & | & \mathsf{N} := \mathsf{e} \\
 & | & \mathsf{Action} \lhd \mathsf{b} \rhd \mathsf{Action} \\
 & | & \mu\, \mathsf{N} \bullet \mathsf{Action} \\
\mathsf{Comm} & ::= & \mathsf{N}\,\mathsf{CParameter}^{*} \\
\mathsf{CParameter} & ::= & ?\mathsf{N} \mid !\mathsf{e} \mid .\mathsf{e}
\end{array}
$$

Figure 2.2: BNF Syntax of Circus Actions

2.2 Introduction to the Available Tools

Community Z Tools – CZT: CZT [2] is an open-source Java framework to build formal method tools for Z and Z dialects. The specifications in LaTeX, Unicode and XML formats can be parsed, typechecked, transformed, animated and printed using the formal method tools included in CZT. The supported languages of the latest version of CZT are Z, Object Z and *Circus*. A limited subset of Z is supported by the animator available in the tool.

The paper [MFMU05] shows how to support new extensions of Z within the CZT framework while minimising the effort required to build a new Z extension. A variety of reuse mechanisms are used in the CZT framework. These mechanisms are: generation of Java code from hierarchical XML schemas; XML templates for shared codes; and several design patterns for maximising reuse of Java code.

Failure Divergence Refinement – FDR: FDR [For05] is a *CSP* based model-checking tool. More precisely, FDR can be described as a refinement checker. Refinement is a term for the process of incremental implementation of the system from the specification. In general, usually it is not possible to construct a program directly from its specification, then prove it to be correct. Instead, the program should be constructed in small steps, each time adding more detail. Since the changes are small, it is relatively easy to prove at each stage that the implementation satisfies the specification. If S is a specification and P is a program then $P \sqsupseteq S$ means that the program P refines the specification S. The FDR tool does refinement checking based on the traces, failures and failures-divergence models.

Saoithin: Saoithin [But10] is a theorem prover designed to support the Unifying Theories of Programming (UTP) framework. It is based on the UTP literature [HH98] so that it can support the proofs containing higher order logic, alphabets and "programs as predicates". It mainly deals with proofs in equational style.

[2] http://czt.sourceforge.net/manual.html

2.3 Unifying Theory of Programming

While mixing two different languages, the unification of the semantics of the languages is a matter of concern. So, there must be a unification framework so that the two worlds of these languages could be mixed together.

The Unifying Theories of Programming (UTP) [HH98] proposes a unification of different programming paradigms based on the theory of relations. The unification allows the exploration of different paradigms. The relation between the paradigms can result in mappings that relate specifications in abstract models to programs in more concrete models; in UTP, the refinement relation is simply a reverse logical implication. The semantics of *Circus* in UTP framework are explained in detail in [OCW09]. Furthermore, the UTP semantics of *CSP* are discussed in [CW04].

2.4 The UTP Semantics of *CSP* and *Circus*

In UTP, the language semantics are described using observation variables with their undashed and dashed versions i.e. initial and final observation variables. Each programming construct is given by a relation between the initial and final observation variables.

1. ok, ok' depict the successful start and termination of a program.
2. tr, tr' depict the initial and final traces (list of events) of a program.
3. ref, ref' depict the initial and final refusal set for a program.
4. $wait, wait'$ depict the waiting status of a program for an interaction with its environment.

The theory being studied has three essential parts:

1. Alphabets: the collection of names for the theory.
2. Signature: the set of syntax rules.
3. Healthiness Conditions: identification of important properties. Each healthiness condition captures an important fact of the computational model of the theory under study.

In design theory of UTP, $P \vdash Q$ depicts a design with pre-condition P and post-condition Q. It is defined as:

$$(P \vdash Q) = (ok \wedge P \Rightarrow ok' \wedge Q)$$

So, the design definition states that if a program starts in a state satisfying P, then it will terminate, and on termination Q will be true.

2.4.1 Healthiness Condition: Reactive Processes

Healthiness Condition	Meaning
$\mathbf{R1}(P) = P \wedge tr \leq tr'$	A process does not change the past history of events.
$\mathbf{R2}(P) = \exists s \bullet P[s, s \frown (tr' - tr)/tr, tr']$	The behaviour of a reactive process is oblivious to what has gone before.
$\mathbf{R3}(P) = II_{rea} \lhd wait \rhd P$	If previous process is unfinished, then P should not start. Here, $II_{rea} \mathrel{\widehat=} DIV \vee ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref$ and $DIV \mathrel{\widehat=} \neg ok \wedge tr \leq tr'$
$\mathbf{R} = \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}$	The composition of above three healthiness conditions is **R**.

2.4.2 Healthiness Conditions: *CSP* Processes

Healthiness Condition	Meaning
$\textbf{CSP1}(P) = P \vee (\neg ok \wedge tr \leq tr')$	The extension of the trace is the only guarantee on divergence.
$\textbf{CSP2}(P) = P;\ J$	A process may not require non-termination.
$\textbf{CSP3}(P) = SKIP;\ P$	A process does not depend on *ref*.
$\textbf{CSP4}(P) = P;\ SKIP$	A terminating process does not restrict ref'.
$\textbf{CSP5}(P) = P \parallel\!\mid SKIP$	**CSP5** states that a deadlocked process that is refusing some events offered by the environment is still deadlocked in an environment that offers even fewer events.

In **CSP2**, $J = (ok \Rightarrow ok') \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref$.

2.4.3 Healthiness Conditions: *Circus* Processes

Healthiness conditions for *Circus* processes are explained below:

$$\textbf{C1}(A) = A;\ Skip$$

This condition is similar to **CSP4** which states that the value of the variable ref' has no relevance after termination.

$$\textbf{C2}(A) = A \, [\![v \mid \emptyset]\!] \, Skip$$

This condition is similar to **CSP5**.

The third healthiness condition for *Circus* processes, **C3** states that the precondition of a process expressed as a reactive design contains no dashed variables.

$$\textbf{C3}(A) = \mathbf{R}(\neg A^f_f;\ true \vdash A^t_f)$$

2.4.4 Alphabets of *Circus* in UTP Semantics

According to [OCW09], the UTP semantics of *Circus* contains the alphabet set of $okay, wait, ref$ and tr with their dashed versions. In order to denote the state variables, the predicate $v' = v$ is used to denote $x'_1 = x_1 \wedge \ldots \wedge x'_n = x_n$.

2.4.5 Guarded Action in *Circus* : An Example

The semantics of *Circus* actions are given in [OCW09]. One out of the *Circus* actions i.e. a guarded action has its semantic explanation provided here for an example.

The guarded action: g & A deadlocks if the guard g is false and performs action A if g is true.

In the definition of the guarded action below, $A[b/ok'][c/wait]$ is abbreviated as A^b_c. So, A^f_f depicts $A[false/okay'][false/wait]$. It means that the observation variables ok' and wait are false. Similarly, $A^t_f = A[true/ok'][false/wait]$, meaning if A is not waiting for a previous process to finish, then it does not diverge.

$$g \& A = \mathbf{R}\left(\left(g \Rightarrow \neg A^f_f\right) \vdash \left(\left(g \wedge A^t_f\right) \vee \left(\neg g \wedge tr' = tr \wedge wait'\right)\right)\right)$$

2.5 Dealing with Z Schemas and Assignment Command in *Circus*

A simplified presentation of the syntax of both *Circus* and *CSP* together was shown on page 4 in figure 1.2. In our work, we shall view a process as a collection of action definitions, along with a designated action called the process main action. The action definitions associate a name and possibly some formal parameters with an action body, whose syntax is basically that of *CSP* processes or *Circus* actions, including calls by name to actions. The *Circus* action syntax covers those of *CSP*, with the addition of an assignment statement and a state-partitioning parallel construct.

As *Circus* descriptions freely mix state information through Z and behavioural aspects through *CSP* actions, so dealing with these two things are of importance. Below, we will describe the handling of Z schemas and assignment commands in previous works.

2.5.1 Z Schemas in *Circus*

The denotational semantics for schema expressions in [Oli05] is based on a conversion rule given in [Cav97] which transforms the schema expressions into specification statements. The schema expression is first normalised. In *Circus*, the Z notations for input (?) and output (!) are syntactic sugar for dashed and undashed variables, respectively. So, the schema contains actual undashed variables (udecl), dashed variables (ddecl') and the predicate explaining the effect of the schema. The definition of schema expression in this work is as follows:

Definition: $[udecl;\ ddecl' \mid pred] \mathrel{\widehat=} ddecl : [\exists ddecl' \bullet pred, pred]$

In [Oli05], the laws of schemas describe the interaction between the Z and *CSP* part of a *Circus* specifications. The functions named *usedV* and *wrtV* are additionally defined to specify the laws. The *usedV* function gives the set of used variables i.e. the variables which are only read, but not written. The *wrtV* function gives the set of variables which are written. Furthermore, in case of schema expressions, the *wrtV* function gives the set of variables that are constrained by the schema. v' is used the describe the list of dashed $v'_1, \ldots, v'_n$ free variables (DFV) of schema expressions *SExp*. Similarly, $v' : T$ notations describe the list of variables with their respective types, defined in the schema expression. The definition for *wrtV* function is given as:

$wrtV(SExp) = \{v' : DFV(SExp) \mid SExp \neq (\exists v' : T \bullet SExp) \land [v, v' : T \mid v' = v] \bullet v\}$

Here, all the dashed free variables are hidden first and then they are declared again. Their types are constrained to be of the same type as defined in the schema. If the schema expression is changed, it means that their actual values have been changed and these should be in the set of written variables.

2.5.2 Dealing of Assignment Command

In [Oli05], for assignment command, the semantics suggest non-divergence resulting a successful termination, leaving the trace unchanged. The assignment sets the final value of the variables in the left hand side to their new values. Remaining values, denoted by u in the definition, remain unchanged. The symbol u is defined as $(u \mathrel{\widehat=} v \setminus \{x_1, \ldots, x_n\})$. The definition for assignment command is as follows:

$x_1, \ldots, x_n := e_1, \ldots, e_n \mathrel{\widehat=} \mathbf{R}(true \vdash tr' = tr \land \neg wait' \land x'_1 = e_1 \land \ldots \land x'_n = e_n \land u' = u)$

2.5.3 Mechanical Abstraction of CSP$_Z$ Processes

In [EL02], a mechanical strategy to transform an infinite CSP$_Z$ process, composed of Z and *CSP* constructs into a form suitable for model-checking, was proposed. To allow the infiniteness of CSP$_Z$ process, two theories are integrated together: one is data independence of the behavioural aspects and second is abstract interpretation of the data structure aspects. The goal defined in the work [EL02] was to propose a strategy for analysis of infinite CSP$_Z$ processes in which the user intervenes only in the theorem proving step. The strategy is based on

the integration of model checking and theorem proving. The strategy was a combination of data independence and abstract interpretation. In the proposed strategy of this work, the authors had considered data independent CSP_Z processes and data dependencies in the Z part using abstract interpretation.

2.6 *CSP* Normal Forms

Chapter 11 of [Ros98] provides information about getting normal forms of *CSP*. **Normalisation**, in general, refers to the process of re-arranging the both sides of an equation to have a particular shape. Likewise, *CSP* also has a normal form. This is required for the use of *CSP* in FDR with better performance. The normal form of *CSP* has the following form:

$$\mathsf{Stop} \mid \mathsf{Skip} \mid \mathsf{Chaos} \mid \sqcap(\Box\ (g\&e \rightarrow NF))$$

This equation depicts that the internal choice of external choices of guarded prefix actions leads to other normal form expressions.
The normal form of *CSP* can be achieved by looking at the laws of *CSP*.

- Non-determinism distributes through everything.
- Step laws allow prefixes to be pulled out of other language constructs.
- Step laws can eliminate parallel and hiding.

For getting a normal form of *CSP*, the start of the process is getting the head normal form “hnf” (Ch 11, p287) [Ros98].

- *DIV* is in hnf.
- *SKIP* is in hnf.
- $?x : A \rightarrow P$ is in hnf.
- $(?x : A \rightarrow P)\ \Box\ SKIP$ is in hnf.
- If each Q_i is a hnf, then $\sqcap_{i=1}^{N}\ Q_i$ is in hnf.

Reducing an arbitrary P into hnf is as follows:

$$\begin{array}{rcl}
hnf(?x : A \rightarrow P) & \widehat{=} & ?x : A \rightarrow P \\
hnf(Q \sqcap R) & \widehat{=} & hnf(Q) \sqcap hnf(R) \\
hnf(STOP) & \widehat{=} & ?x : \{\} \rightarrow P \\
hnf(\mu N \bullet F(N)) & \widehat{=} & hnf(\mu\, unwind(\mu N \bullet F(N))) \\
hnf(Q \oplus R) & \widehat{=} & \sqcap_{i=1}^{N}\ (?x : A_i \rightarrow hnf(Q)) \oplus (?x : B_i \rightarrow hnf(R))
\end{array}$$

2.6.1 A Similar Work on Normalisation of *CSP* Processes

The paper [BM10] provides the information on the unfolding process for *CSP* operators. *Unfolding* is a special case of parametric recursion and unfolding of these processes i.e. $\Box$ and $|||$ is achieved through reduction to a normal form. [BM10] demonstrated the complementary features of *CSP* and bisimilarity features of CCS.

CSP defines a failures preorder which combines specification and implementation in a single syntactic framework. CCS defines *process* as an equivalence class keeping the order in which the non-deterministic choices are made. If bisimilarity is shown by $\sim$ and we have $P_1 \sim Q_1$ and $P_2 \sim Q_2$ and there is a *CSP* operator *OP*

then $OP(P_1,Q_1) \sim OP(P_2,Q_2)$. The operators defined in [Hoa04] not only worked for *failures* preorders but also satisfied the bisimilarity equivalence. As a result, the theory of CCS and *CSP* can be combined. The equivalence introduced by unfolding is closely related to bisimilarity. Roscoe's *CSP* [Ros98] defined head normal forms. Two processes are *failures' equivalent* if their expressions in normal forms are equivalent.

Unfolding of the *CSP* operators in [BM10]:

If the symbol $\hookrightarrow$ stands for unfolding operator, then the following are the equations for unfolding of the *CSP* operators suggested by [BM10]:

$$F \parallel G \hookrightarrow H \text{ where } H(z) = \left\{ \begin{array}{ll} F(z) \parallel G & (z \in X \backslash Y) \\ F \parallel G(z) & (z \in X \backslash Y) \\ F(z) \parallel G(z) & (z \in X \cap Y) \end{array} \right\} \text{ and } H(\tau) = F(\tau) \parallel G \cup F \parallel G(\tau)$$

$$F \Box H \hookrightarrow H \text{ where } H(x) = F(x) \cup G(x),\ (x \in X) \text{ and } H(\tau) = F(\tau) \Box G \cup F \Box G(\tau)$$

$$F \mid\mid\mid G \hookrightarrow H \text{ where } H(\mu) = F(\mu) \mid\mid\mid G \cup F \mid\mid\mid G(\mu),\ (\mu \in X \cup \{\tau\})$$

$$F \setminus Z \hookrightarrow H \text{ where } \begin{array}{l} H(x) = F(x) \setminus Z,\ (x \in X \setminus Z) \\ \text{and } H(\tau) = F(\tau) \setminus Z \cup \bigcup_{x \in X \cap Z} F(x) \setminus Z \end{array}$$

$$F \sqcap G \hookrightarrow H \text{ where } H(\mu) = F(\mu) \cup G(\mu)\ (\mu \in X \cup \{\tau\})$$

2.7 Summary

Here, we conclude this chapter and summarise what has been covered in this chapter. We started with a brief introduction to the Z notation, *CSP* and *Circus*. The basic structure of *CSP* and *Circus* programs was elaborated through an example process definition. Then, a key subset of *Circus* and *CSP* abstract syntax trees were described. A brief introduction to the available tools for the target languages was given. Then, we introduced the Unifying Theories of Programming (UTP) framework and described how the semantics of programs can be described through this framework. Here, the observation variables and healthiness conditions of *Circus* and *CSP* were introduced. Then, very specific topics related to this research were introduced, for example, handling Z schemas and assignment commands in *Circus*, and the description of *CSP* normal forms. The following chapter gives state of the art of the thesis.

Chapter 3

State of the Art

As the name of the chapter suggests, this chapter describes the state of the art section of our work. First, it describes the similar work available in the literature. Then it suggests what we are proposing in this thesis. Later on, it gives a whole picture of some of the interesting work related to state-rich *Circus*. In the last section of the chapter, we describe possible case-study areas for our work.

3.1 Related Work

The paper [OC04], also described in Chapter 6 of the PhD of M. Oliveira [Oli05] suggested a tool named JCircus. This work provided the translation rules for automatically generating the Java code from a *Circus* specification. The translation strategy was suggested on the basis of a JCSP library, providing the implementation of *CSP* constructs of *Circus*. In [OC04] and Chapter 6 of [Oli05], the implementation of multi-synchronisation on a channel was provided by a class `GeneralChannel`.

Simple Example of a Translation Rule

In case of a Java program, the rule for internal choice is to use `RandomGenerator` class to produce a pseudo-random number, which is compared in a switch statement to pick a random process.

$[\![Proc_1 \sqcap \ldots \sqcap Proc_n]\!]^{Proc} =$

```
int choosen = RandomGenerator.generateNumber(1,n);
switch(choosen) {
    case 1: { [[Proc_1]]^Proc } break;
    ...
    case n: { [[Proc_n]]^Proc } break;
}
```

There are around forty translation rules suggested in [Oli05] to cover a significant amount of *Circus* to transform into their Java versions. These translation rules were implemented in [dF05]. This work ended up in the first version of JCircus. The tool had a simple graphical user interface. The *Circus* specification written in LaTeX was given as input by specifying the folder containing it. The next two fields specified the project place holder where the output, generated Java code, should be placed. By hitting the translate button, the output project folder was generated by the translator. The work presented in the paper [dF05] was the starting point of our work of implementing the translation from *Circus* to CSP_M as this provided us the basis for parsing *Circus* specifications.

The basic architecture of JCircus is shown in figure 3.1. JCircus contains three main modules. The parser is the first one, which receives a LaTeX file containing the specification, parses it, and creates the AST that represents the specification. The AST is given as input to the type checker, which performs type inference, checks for type errors, and annotates the AST nodes for expressions with their types. The third module is the translator, which is the contribution of [dF05]. The `TranslatorVisitor` class contains all the methods to convert *Circus* processes to their Java equivalent.

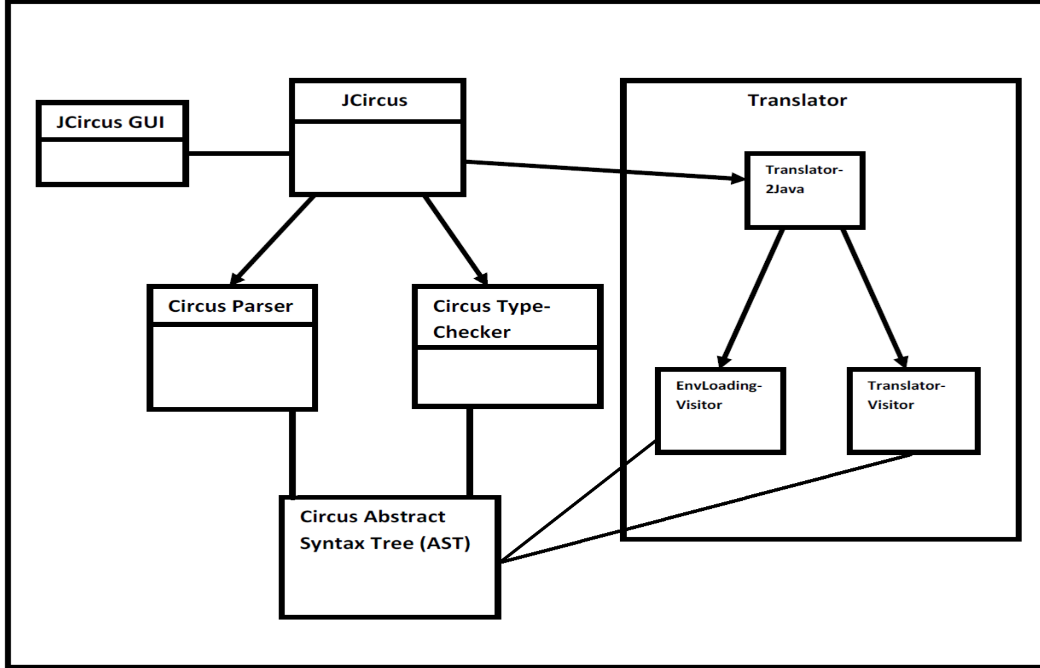


Figure 3.1: The basic architecture of JCircus classes. Figure 4.4 of [dF05]

The second version of JCircus, named JCircus 2.0 was suggested in [BO12]. The difference between JCSP and the *Circus* implementation of concurrency had imposed restrictions on the translation strategy in [Oli05] and, as a result in the developed tool JCircus. The improvements in JCircus 2.0 listed by the authors of [BO12] are: the improved translation strategy to allow multiway synchronisation; complex structure of communications on channels and translation involving mix of parallel and interleaving constructs. The performance analysis between the first version and second one is also provided in [BO12].

In our work, to establish the link between a *Circus* based tool and a *CSP* based tool, the specifications written in one domain are to be translated in to the other.

Let us take an example of a simple process, by considering figure 1.3 shown on page 4. In the case of a *Circus* process, `x` is assigned value in the specification. While doing the same in CSP_M, the value is passed as a parameter. In the case of this particular example, we need to extend UTP semantics to cover parameteric processes.

The main *difference* between previous works and ours is that others translate the variables into processes with set and get channels [CH13, Sch99], while in our approach, we turn variables into parameters.

Translating from *Circus* to *CSP* requires finding a way to model the effect of the variable assignment of the former in the latter. One possible approach is to model each variable as a process with get and set channels [CH13, Sch99] e.g.:

$$\begin{aligned} Var(n,v) \;\widehat{=}\;& get.n.v \rightarrow Var(n,v) \\ & \Box set.n.v' \rightarrow Var(n,v') \end{aligned}$$

However, every expression in $x := e$ has to be rendered as a process that performs an appropriate sequence of

gets to evaluate e, to v (say), and then performs a $set.x.v$ event. In addition, the interpretation of $A\ {}_L||_R\ B$ is, that A may only modify variables in L, which is disjoint from R, which itself determines what B is allowed to modify. Both A and B can read all variables, but cannot see any modifications made by the other process. Operationally, A and B run on local copies of the starting variable state, and the changes each has made only get merged together when both have terminated. This would require us to have explicit state-copying and merging processes acting as bookends to every parallel construct.

Instead, we are going to adopt the approach of turning assigment variables into parameters of action definitions, so a *Circus* fragment like

$$A \mathrel{\widehat=} x := y + 1 \mathbin{;} B$$

is translated to

$$A(x,y) \mathrel{\widehat=} B(y+1,y).$$

Now, we describe some of the literature which contains similar efforts in the past.

In [YP07], the automatic translation from 'Combined B' and *CSP* to Java was proposed. The authors proposed the translation rules of the given specifications to multi-threaded Java. This was achieved using the extended JCSP channel classes. A prototype tool was presented. This tool was an extension of the ProB tool. [YP07] gave a working example and then the formal verification of the translation was provided.

In [PS04], a tool for conversion of a subset of *CSP* to Handel-C code is presented. Handel-C is a C like language and is portable to produce program files to upload on FPGAs. The output script is in a functional language style. This tool is not successful for every script of *CSP* but two working examples were provided in the paper which were successfully being loaded on FPGAs. In short, the contribution of [PS04] was the automatic generation of a Hardware Description Language (HDL) from a *CSP* program.

In [Bro08], special types called monads are used to express the concurrent *CSP* processes within Haskell - a lazy functional programming language. This resulted in a new *CSP* library for Haskell known as Communicating Haskell Processes (CHP). In [Bro09], the authors presented the strategy to generate *CSP* models of Haskell implementations. These Haskell implementations are in the CHP language, proposed in [Bro08].

In [BH99, BH01], an Occam-based programming language was used for the automatic compilation of high-level programs to low-level circuits. Also, the compilation process was verified which ensured its correctness. The compiler was written in Prolog. The translation used for assignments was to convert then into a hardware latch with a delay.

In [VG08], descriptions of concurrent systems written in the RAISE Specification Language were translated to CSP_M. The need for the translation was to use the FDR toolkit to model-check RSL specifications. In [VG08], it was shown that this translation is a strong bisimulation. Properties of RSL specifications could be inferred using FDR generated results.

In [TG08], a translator was developed to convert RSL specifications to CSP_M. Overview of semantic and syntactic differences between two languages were discussed. A translation subset was defined, limited to only those constructs in RSL that matched those in *CSP*.

Now, we mention some of the other literature related to *Circus* and CSP_M.

Paper [BSW07] presents a generic framework of UTP theories for describing systems whose behaviour is characterised by regular time-slots, compatible with the general structure of the Circus language. The slotted-Circus framework in [BSW07] is parameterised by the particular way in which event histories are observable within a time-slot, and specifies what laws a desired parameterisation must obey in order for a satisfactory theory to emerge. Papers [GB09b, GB09a] are related to an ongoing work that describes a complete denotational semantics, in the UTP framework, of slotted-Circus. Slotted-Circus is a generic framework for reasoning about

discrete timed / synchronously clocked systems. The key result presented in [GB09b] is a comprehensive semantics of the entire language that addresses various semantics issues, This work laid foundations for its future extensions, particularly towards prioritized choice.

In [OCW06], the authors mechanised the combination of different programming paradigms in a theorem prover, ProofPower-Z. This work included the theories of alphabetised relations, designs, reactive and *CSP* processes. The authors targeted *Circus* for the mechanisation of the unification theory [HH98]. In the paper [SCS06], object-oriented programming concepts were studied in languages like Java and C++ within the UTP framework. By using the combination and extension of theories of designs and higher-order procedures, the authors described subtyping, data inheritance, (mutually) recursive methods, and dynamic binding in the UTP. Paper [CHW06] described a theory of pointers and records providing a representation for objects and sharing in languages like Java and C++. The authors defined the theory as a restriction of the general theory of relations, in which terminating and non-terminating programs are not distinguished from each other. So, the authors linked it with the theory of designs. This provided a basis for reasoning about the correctness of pointer-based sequential programs. The work in [CHW06] gave the semantics of an object-oriented language. Here the constructs for specification of state-rich concurrent systems were integrated.

The work [BGW09] investigated the interactions between program variable state visibility and communication behaviour in state-rich *CSP*-like processes, using the UTP framework. This gave a result that, if the variable state is visible during the wait state of communication, then ordinary programs put together in a normal way ended up with miraculous behaviour – to be avoided in a theory of feasible programming.

3.2 Possible Case Study Area

The description of background reading in the following subsection belongs to the possible case study area of my PhD work.

3.2.1 Verification of Flash Memory Behaviour

After the start of the 'Grand Challenge in Computing' (GC6) [HM05] project with a special focus on mission critical filestores, a number of efforts have been made to formalise flash memory and filestores. Paper [BW07] gave the Z notation for a formal model of NAND flash memory. The model describes the internal architecture of NAND flash memory with some abstractions. Paper [BFW09] was a step ahead towards mechanising the formal model of NAND flash memory. The Z/Eves Theorem Prover has been used for describing the state model and initialisation operation of NAND flash memory. Papers [Cat08, BOC09] are about modelling the flash memory behaviour using *CSP*. In these works, Open NAND Flash Interface (ONFi) specifications were modelled. Instead of writing *CSP* directly, the ONFi's finite state machines' specifications were converted into intermediate form using State Chart XML (SC–XML). This XML was then automatically converted into *CSP* via XML Transforms (XSLT).

Paper [MAF08] reports on the use of Alloy and HOL (a theorem prover) to validate and verify a VDM model of the Intel Flash File System Core specification, as a part of the 'Verifiable File System' (VFS) project[1]. Paper [KJ08] describes the formal modeling and analysis of a design for flash-based filesystem in Alloy. The authors modelled the basic operations of filesystem as well as other features that are crucial to NAND flash hardware, such as wear-leveling and erase-unit reclamation. Papers [KCKK08b, KCKK08a, KK09, KKK08] document experiments in the formal verification of OneNAND^{TM} Flash Memory, which is a trademark of Samsung Electronics. Flash memory is an essential part of mobile devices, so in order to operate mobile devices successfully, it is essential that flash memory be controlled correctly through the device driver software. In [KCKK08b, KCKK08a, KK09, KKK08], Kim et. al. formalized the multi-sector read operation of OneNAND

[1] *http://wiki.di.uminho.pt/twiki/bin/view/Research/VFS/WebHome*

flash memory using NuSMV, Spin, and CBMC model checkers to verify the correctness of the read operation. Results obtained from model checking techniques demonstrate the feasibility of using these techniques to verify the control algorithm of device driver. Furthermore, Concolic (**Conc**rete + Symb**olic**) testing was applied for the verification of the Multi-Sector Read Operation in the flash control software.

Flash Device Behaviour Modelling Extension

As an intial case study and to have hands-on experience on CSP_M and the model checker FDR, *CSP* models for flash device behaviour presented in [Cat08, BOC09] were extended. This work has been published in conference proceedings [BB10b]. The body of the paper is included in chapter 4. The achievements were:

1. Upgrading the Open Nand Flash Model (ONFi 1.0) *CSP* sources to ONFi 2.1.

2. Tests done on the model with more rigorous testing using full Failure-Divergence Refinement. In the old version, the testing was done with the weaker testing technique of trace refinement only.

3. By pushing the use of hiding deeper into the model, as well as the compression techniques available, the state-space of the model was much reduced.

3.2.2 Formal Verification of Cache Coherency Protocols

In the distributed and shared memory systems, the performance of the system is very dependent on the 'coherency' of cache memories. Cache memory's purpose is to place or access the most frequently used data through the cache instead of accessing the data from the main memory each time. In shared and distributed memory architectures, a number of caches along with a shared bus are present. As a consequence, the need for 'coherent' data placed in each cache is obvious so that most updated copy of data is available to every processor. This problem is resolved through 'Cache Coherency Protocols'. The correctness of these protocols is of key importance. The papers [ARA04, CMP04, CMW90, DNA05, EK03, FS96, NG02, PD97, QGPY08, She00, SPH$^+$00, SD95, WE04] are some previous works about proving the correctness of cache coherence protocols.

Use of Simple*Circus* for Cache Coherency Protocol Specification

The notation of Simple*Circus* is used to specify a cache coherence protocol given in [FS96] on page 89, section 9.2. This provided the future use of the Simple*Circus* notation for such protocols' specification.

3.2.3 Formal Verification of Hardware Protocol – AMBA Bus Protocol

For high performance data transfers among the IP cores in the System-on-chip (SoC) designs, different bus protocols are used. The work [RMK03] is a case study in the formal verification of a SoC bus protocol named the Advanced Micro-controller Bus Architecture (AMBA) protocol from ARM. The particular emphasis of this work was the formal specification of the AMBA protocol. The crucial design invariants were verified using the SMV model checker. Pipelining and split transfer in the AMBA protocol gave scenarios which could not be analysed with informal ways of reasoning. Roychoudhury et al [RMK03] detected a potential bus starvation scenario in the AMBA protocol.

3.2.4 Formal Verification of Networking Protocols

An interesting work in the formal verification of networking protocols is [ISK06]. A portion of the Dynamic Host Configuration Protocol (DHCP) was modelled and verified in [ISK06]. The selected part of DHCP was the

assignment of new IP addresses to newly arriving hosts. For modelling and verification PROMELA (PROcess Meta LANguage) was used in SPIN. SPIN provides verification capability for a number of communication protocols either by C program generation or by random simulations. In [ISK06], analysis and verification of deadlock, livelock-freedom, and improper termination of the DHCP protocol under constraints of message loss and arbitrary errors was done. The much needed properties of the protocol were also verified using linear temporal logic (LTL). The properties verified were message integrity and conflict handling in allocated IP addresses. The first property was proved error-free while the second property was violated. Another interesting work is [BOG02] in which authors used HOL, an interactive theorem prover, with SPIN to prove key properties of a distance vector routing protocol.

3.3 Summary

This chapter described the state of the art of our thesis. An important point made here was the selection of our approach of turning variables into parameters, while others translate the variables into processes with set and get channels or clocked hardware. Then, we gave a detailed overview of the related work available in the literature. Then, a possible case study area for our work was discussed in detail. We chose here a flash device behaviour CSP_M model. This resulted in an initial case study, which is the topic of the next chapter.

Chapter 4

Flash Work – An Initial Industrial Case Study

The material in this chapter is based on [BB10b], published and presented at the International Conference on Frontiers of Information Technology, 2010, held at Islamabad, Pakistan. In section 3.2.1 on page 22, we described the possible case study areas for our work. In initial stage of our research, the case study mentioned in this chapter was carried out. This provided us hands-on experience with the Failures-Divergence Refinement (FDR) toolkit – a model checker for *CSP*.

4.1 Introduction

We present our experience of working with the Failures-Divergence Refinement (FDR) toolkit while extending our modelling of the behaviour of Flash Memory. This effort is a step towards the low-level modelling of data-storage technology that is the target of the POSIX filestore mini-challenge. The key objective was to advance previous work presented in [Cat08, BOC09] to cover the full Open Nand-Flash Interface (ONFi) 2.1 model. The previous work covered a sub-model of the mandatory features of ONFi 1.0. The FDR toolkit was used for refinement/model-checking. In addition to the compression techniques available in FDR, we also experimented with FDR Explorer - an application-programming interface (API) that allowed us to get a better picture of FDR performance. This paper summarises the progress we made, and the limits we encountered. We are now able to verify many of the operations in ONFi 2.1 model using full Failures-Divergence refinement checking, rather than just trace refinement. Through the use of compression techniques available in the FDR toolkit and in particular by hiding the events deeper in the model, we were able to get compression of the state-space. The work also reports the number of attempts to compile the full ONFi 2.1 model.

The "Grand Challenge in Computing" [Hoa03] on Verified Software [Woo06, HMLS09], has a stream focussing on mission-critical filestores, as required, for example, in space-probe missions [JH05]. Of particular interest are filestores based on the NAND Flash Memory technology, very popular in portable datastorage devices such as MP3 players and datakeys.

This paper follows on from initial formal models of NAND Flash Memory, reported in [BW07] and then [BOC09] based on the specification published by the "Open NAND Flash Interface (ONFi)" consortium [$H^{+}06$]. The first two works looked at the formal model of flash memory in terms of its internal data storage architecture, and the top-level operations that manipulate that storage.

The work in [BOC09] reports on modelling and analysing the finite-state machines in [$H^{+}06$] that describe the internal behaviour of flash devices. The modelling was done using machine-readable Communicating Sequential Processes (CSP_M) [Ros98] and the FDR2 tool [For05] for the analysis, and was reported in detail

in an M.Sc dissertation [Cat08]. The works [Cat08, BOC09] also describe a methodology for model data-entry based on the "state-chart" dialect of XML (SCXML) using XSLT to translate into *CSP*. Using XSLT to convert the intermediate XML to *CSP* saved time and reduced error-proneness in the semi-automatically generated *CSP* code. The key objective of recent work was to advance the work presented in [Cat08, BOC09] to cover the full ONFi 2.1 model [H$^+$09] and to get stronger and more complete results from FDR.

In the next section (§4.2) we describe the relevant aspects of ONFi flash devices, and look at related work (§4.3). We then proceed to present the development of the *CSP* model (§4.4), the analyses performed with it (§4.5) – main contribution lies in this section, and conclude (§4.6).

4.2 Background

A flash memory device is best viewed as a hierarchy of nested arrays of bytes/words, plus additional state and storage facilities at various levels. At the bottom we have *pages*, arrays of bytes, which comprise the basic unit for both writing (programming) and reading (operations *PageProgram* and *Read*). The next level up is the *block*, an array of pages, that is the smallest level at which erasure (operation *BlockErase*) can take place. Blocks are aggregated together under the control of a *logical unit (LUN)*, which is the smallest entity capable of independent (concurrent) execution. A LUN also has one or more local registers the same size as a page (*page-registers*), used as temporary storage when transferring data to/from block pages, and a *status register* recording key information about ongoing operations, or those just completed. The status register has 8 bits, of which only bit 6 (a.k.a "SR[6]"), is of interest, used to indicate the ready/busy status of a LUN. LUNs are collected together into *targets*, which have their own means of communication off-chip. A physical flash memory chip (or *device*) may have several targets, depending on the number of available I/O pins. This paper focusses on the target level and below, with a particular emphasis on the interactions between LUNs and their containing target.

4.2.1 Flash Memory Operations

The ONFi standard defines a collection of operations that are to be supported by flash devices. Some of the operations are mandatory and must be provided in any ONFi-compliant implementation. The operations, Read, PageProgram, BlockErase and ReadStatus, have already been introduced. The other operations include: *Change…Column* operations that support access to part of a page; *Reset* to allow software to reset a device; *WriteProtect* to direct LUNs to be locked/unlocked against changes; and *ReadID* and *ReadParameterPage* that return data specific to a device such as manufacturer's name, and sizing information.

Other optional operations are also specified, typically providing enhanced performance-improving features that exploit the parallelism provided by the LUNs — in ONFi2.1 there are about 17 of these so we do not list them here. Keeping the size of the model in mind, our *CSP* model comes in two versions, one covering only the mandatory behaviour, whilst the other also includes the optional operations.

4.2.2 Host-Target Communication

We use the term *host* to refer to any entity interacting with a flash memory device. Most communication between a host and target is mediated through a single bi-directional byte-wide I/O port, so the hardware interface is essentially serial. Conceptually, four types of transfer take place across this port: Command Write $CW(opcode)$, a single byte denoting a command is sent by the host to the target; Address Write $AW(addr)$, a byte denoting part of an address is sent to the target; Data Write $DW(byte)$, a data-byte is sent to the target; and Data Read $DR(byte)$, A data-byte is received from the target.

Executing a typical operation involves a series of transfers of the four types listed above, typically with some

T_RPP_ReadParams	The target performs the following actions: 1. Request LUN tLunSelected clear SR[6] to zero. 2. R/B# is cleared to zero. 3. Request LUN tLunSelected make parameter page data available in page register. 4. tReturnState set to T_RPP_ReadParams.	
1. Read of page complete	→	T_RPP_Complete
2. Command cycle 70h (Read Status) received	→	T_RS_Execute
3. Read request received and tbStatusOut set to TRUE	→	T_Idle_Rd_Status

Figure 4.1: ONFi Target State Example [H+09], Page 175.

```
<state id="T_RPP_ReadParams">
    <onentry>
        <assign location="isReadyBusy" expr="false"/>
        <event name="tl.tLunSelected!targRequest"/>
        <event name="tl.tLunSelected!retrieveParameters"/>
        <assign location="tReturnState" expr="T_RPP_ReadParams"/>
    </onentry>
    <transition event="tl_sync.tLunSelected -&gt; tl.tLunSelected.readPageComplete" target="T_RPP_Complete"/>
    <transition event="ht_ioCmd.cmd70h" target="T_RS_Execute"/>
    <transition event="ht_read" cond="tbStatusOut==true" target="T_Idle_Rd_Status"/>
</state>
```

Figure 4.2: ONFi Target State Example – SC-XML Code

waiting inbetween. For example, a *Read* operation involves the following (typical) initial series of transfers:

$$CW(readOpcode);\ AW(addr_4);\ \ldots;\ AW(addr_0);\ CW(confirm)$$

The host has then to wait whilst the addressed data is pulled from the relevant page into the selected LUN's page-register, as signalled by the LUN status register. LUN status can be read either directly via an output pin ("hardware" status) or by performing a *ReadStatus* operation ("software" status).

4.2.3 The ONFi state machines

The internal behaviour of ONFi devices is described by two finite-state machines (FSMs) [H+09, §7], one describing the behaviour of a target, the other capturing the actions of a LUN. An example state entry, for the target state `T_RPP_ReadParams` (for the *ReadParameterPage* operation) is shown in figure 4.1. The box at on the top-right describes the events that occur on entry to the state. The three rows below describe the subsequent conditional behaviour in this state. The left of each row describes a input event or condition whilst the right indicates the resulting state transition, with the conditions being evaluated in the order in which they appear.

In figure 4.2, we show the SC-XML code for `T_RPP_ReadParams` state. In figure 4.3, we show the generated HTML for `T_RPP_ReadParams` state. In figure 4.4, we show the generated CSP_M for `T_RPP_ReadParams` state. Here, it can be noted, that in CSP_M, setting one parameter requires the whole parameter list to attach to the calling action. Whereas, doing the same in *Circus* is much easier. Only a single assignment command sets the value of the state variable.

4.3 Related Work

Formal model-checking techniques have been applied to the verification of the Samsung OneNAND flash device driver [KCKK08b], with particular emphasis on a multi-sector read operation implemented within the FTL. The model-checkers explored were NuSMV, Spin and CBMC. The best tool was reported as CBMC[CKL04], a SAT-solver based model-checker, that works directly with C source code. Follow-on work [KK09] described the use of a concolic testing method applied to the multi-sector read operation for the flash memory. This

T_RPP_ReadParams	1. Event: tl.tLunSelected!targRequest 2. Event: tl_setSR6.tLunSelected!false 3. isReadyBusy set to false 4. Event: tl.tLunSelected!targRequest 5. Event: tl.tLunSelected!retrieveParameters 6. tReturnState set to T_RPP_ReadParams		
	1. tl.tLunSelected readPageComplete	->	T_RPP_Complete
	2. ht_ioCmd.cmd70h	->	T_RS_Execute
	3. ht_read (if tbStatusOut==true)	->	T_Idle_Rd_Status

Figure 4.3: ONFi Target State Example – Generated HTML from SC-XML

```
T_RPP_READPARAMS(tbStatusOut,tbChgCol,tCopyback,tLunSelected,tLastCmd,tReturnState,
     tbStatus78hReq,cmd,isReadyBusy,isWriteProtected,dataBit,addrReceived,lun0ready,
    lun1ready,intCounter,addr3Block,addr2Page,addr1ColH,addr0ColL) =
        tl.tLunSelected!targRequest -> tl_setSR6.tLunSelected!false ->
        tl.tLunSelected!targRequest -> tl.tLunSelected!retrieveParameters ->
                (tl.tLunSelected.readPageComplete -> T_RPP_COMPLETE(tbStatusOut,
                    tbChgCol,tCopyback,tLunSelected,tLastCmd,T_RPP_ReadParams,
                    tbStatus78hReq,cmd,false,isWriteProtected,dataBit,addrReceived,
                    lun0ready,lun1ready,intCounter,addr3Block,addr2Page,addr1ColH,
                    addr0ColL)
                    []
                ht_ioCmd.cmd70h -> T_RS_EXECUTE(tbStatusOut,tbChgCol,tCopyback,
                    tLunSelected,tLastCmd,T_RPP_ReadParams,tbStatus78hReq,cmd,false,
                    isWriteProtected,dataBit,addrReceived,lun0ready,lun1ready,
                    intCounter,addr3Block,addr2Page,addr1ColH,addr0ColL)
                    []
                 (tbStatusOut==true)
                   & (ht_read -> T_IDLE_RD_STATUS(tbStatusOut,tbChgCol,tCopyback,
                    tLunSelected,tLastCmd,T_RPP_ReadParams,tbStatus78hReq,cmd,false,
                    isWriteProtected,dataBit,addrReceived,lun0ready,lun1ready,
                    intCounter,addr3Block,addr2Page,addr1ColH,addr0ColL)))
```

Figure 4.4: ONFi Target State Example – Generated CSP_M from SC-XML

method combines a concrete dynamic analysis and a symbolic static analysis to automatically generate test cases and an accordingly exhaustive path testing was performed. Furthermore, the authors compared concolic testing method with other model checking techniques applied to the flash file system domain.

A fully automatic analysis, using Alloy, of a flash filesystem is described in [KJ08, KJ09]. This was built on top of a simple flash model (at roughly the same level of abstraction as [BW07]). The basic file operations as well as crucial design features, such as wear leveling and erase-unit reclamation, of NAND flash memory were included in the design. This design also includes a recovery mechanism for unexpected hardware failures. The design was analysed by checking trace inclusion of the flash file system against a POSIX-compliant abstract file system. Similar work, but very much a tools-integration approach to modelling (VDM/HOL/Alloy), was reported in [FSO08, FO09]. The key issue here was matching specific tools to specific verification tasks, and the need to translate between tool notations, in order to have a complete formal verification lifecycle. VDM was used as the main modelling tool, with Alloy and HOL called upon to verify proof obligations that arose.

4.4 The *CSP* Model

The main objective of this and previous efforts [Cat08, BOC09] was to formalise the Target/LUN FSM descriptions in machine-readable *CSP* and then use this as a basis for checking their correctness using the FDR2 refinement checker [For05].

The main criteria for correctness was that the behaviours possible for the interconnected FSMs was consistent with the behaviour patterns for the operations mandated by that same standard.

The state machine notation of the ONFi specification allows for a relatively direct conversion into *CSP*: there is a close correspondence between ONFi states and *CSP* processes. The separation of target from LUNs also echoes the parallel composition features of *CSP*. Multiple LUN processes can be interleaved: required to

synchronize on events with the target, but not with each other. The target-LUN communication events (*TLEvts*) are then hidden and this is put in parallel with a *HOST* process that models the behaviour of the environment that communicates with the flash device. In *CSP* notation this is written (for a single target and two LUNs) as:

$$SYSTEM \mathrel{\widehat{=}} HOST \parallel ((TARGET \parallel (LUN(0) \interleave LUN(1))) \setminus TLEvts)$$

Modelling the communication between host and target was straightforward as this is well documented as the external interface of ONFi devices, and had already been modelled in Z at an abstract level[BW07]. In CSP_M, we used events with names of the form `ht_XXXX` to model these communications, which basically consisted of the byte-level transfers of commands, addresses, data and the single-bit signals (e.g. write-protect input, ready/busy output).

Certain abstractions and simplifications had to be made so that the FDR2 model-checker could perform analysis without running out of memory. So, most data and address items were modelled as single bits, while the command datatype was restricted to the set of known command types, rather than being a full byte. An exception is the *column address* (address of byte within page), which was modelled as two bits to support the *ChangeXXXXColumn* operations.

The 8 state variables of the target had also to be abstracted, and augmented with implicit state data, such as the state of the write protect pin, and the data and address information temporarily in transit, as well as a counter for the number of address chunks expected. This resulted in the addition of a further 13 state components. A similar exercise in augmenting the state had to be done for the LUN FSM as well, to a lesser degree (9 ONFi variables were augmented by a further 3).

4.4.1 *CSP* Data-Entry

Instead of writing *CSP* directly, the ONFi finite state machines specifications are described using Statechart XML (SC–XML). This was then automatically converted into *CSP* via XML Transforms (XSLT) as described in [Cat08, BOC09]. The model has two versions: the full and mandatory version, covering all the operations and only the mandatory ones respectively. The auto-generated *CSP* files for the host, target and LUN state-machines vary between the full and mandatory specifications. The other *CSP* files are hand-crafted and do not vary with the model version. These files are: declarations of datatypes and *CSP* channels (`header.csp`); status register implementation (`SR6.csp`); internal LUN behaviour (`lun-innards.csp`); and combining all the processes to describe various systems (`footer.csp`). At the top-level, we include all the above files in `ONFI.csp`, (or use `ONFI-mandatory.csp` if only the mandatory model is required).

4.5 Model Analysis

The model analysis fell conceptually into two phases: the first focussed on debugging and validating the model, to ensure that it captured the intent of the ONFi specification. The second phase was using the model to analyse the consistency of the entire ONFi document. The model validation is described in detail in [BOC09, §5.1]. The verification process undertaken for the FSMs of ONFi 1.0 document is described in [BOC09, §5.2].

4.5.1 Moving from ONFi 1.0 to ONFi 2.1

First of all, the pre-existing SCXML files of ONFi 1.0 model were updated according to the ONFi 2.1 document [H^+09], after noting the differences between the two versions. This step was straightforward. The comparison of state variables and state entries of the two models are in table 4.1. This clearly indicates that the ONFi 2.1 model is bigger than the previous version.

Description	ONFi 1.0	ONFi 2.1
Target FSM state variables	7	8
Target FSM state entries	77	88
LUN FSM state variables	8	9
LUN FSM state entries	62	68

Table 4.1: Comparison of State Variables and Entries for Two Models

Description	ONFi 1.0	ONFi 2.1	Increment Factor
Transitions in ISM	4490300	7023100	156.4%
States Refined	32,338	47,787	147.7%
Transitions during Refinement	78,469	117,473	149.7%

Table 4.2: Comparison of State Space of Two Models reported by FDR

4.5.2 Running the Model in FDR

After conversion, we initially tried to check the model on a Core Duo machine with processor speeds of 2.00GHz and 1.06GHz and 1.75GB of RAM running Ubuntu Linux, and then on a machine having Core 2 Duo Processors of 2.66GHz and 4.00GB of RAM also running Ubuntu Linux. But in each case FDR stopped during its compilation process and halted all the processes running on the CPU. The model ran successfully on a Quad Processor UltraSPARC-IIIi machine with processor speeds of 1.28GHz and 16GB RAM under Solaris. After this experimentation, all the tests were run on this machine. In addition to this, in order to compile the indexed state machines (ISMs) of the model in FDR2, we had to increase the stack size using the command `ulimit -s 262144`. With these settings we were able to compile and verify all required properties on the mandatory-only version of the model. However, all attempts to handle the full model failed, with a compile-time failure. We now describe various attempts made to get the full ONFi model to run through FDR.

4.5.3 Initial Checks Performed on the Model

The host model in which the status check is done through software was setup as follows:

```
TARGET_TWOLUNS = TARGET [| tl_events |]
            (LUN(lun0) ||| LUN(lun1))
HOST_SW_TARGET_TWOLUNS = INITAL_HS_POWERON
            [| ht_sw_events |] TARGET_TWOLUNS
```

In the case of mandatory ONFi 2.1, when checking the
`HOST_SW_TARGET_TWOLUNS` process for deadlock freedom using failures refinement, the comparison of state space of two models reported by FDR is shown in table 4.2. We see an increase of about 50% in all model-checking size measures reported by the tool.

4.5.4 More Concrete Tests through Failures Refinement Checks on the Model

In previous work the implementations of Read, Page Parameter, MultiRead and Block Erase operations were tested against their specifications using *CSP*'s Traces model. The implementations of these operations were now tailored so that we could do refinement checks in the more powerful Failures model. As all our models were shown to be divergence-free initially, we did not need to perform full Failures-Divergences refinement checks. For these, the implementation of a process looped back to its specification. For example, the *Read* operation was checked as follows.

In the READ_SPEC process we took the HOST_SW_TARGET_TWOLUNS process, hid all the events except the host-target read-related commands and data transfers. The timing diagram of these commands and data transfers are specified on Page 127 of [H+09]. The POWERON behaviour is specified as a sequence of first a reset command (FFh) followed by a read status command (70h). The implementation of the Read operation was defined as a process that performed an expected sequences of host target protocol events for a Read (preceded by a POWERON behaviour).

```
READ_SPEC = HOST_SW_TARGET_TWOLUNS
       \ diff(Events,union
       ({ht_ioCmd.cmds |
       cmds <-{cmd30h,cmd00h,cmd70h,cmdFFh}}
       ,{|ht_ioDataOut|}))
POWERON = ht_ioCmd.cmdFFh -> ht_ioCmd.cmd70h
       -> ht_ioDataOut.true
       -> SKIP -- poweron events
READ_F_IMPL0 = POWERON;
    ht_ioCmd.cmd00h -> ht_ioCmd.cmd30h
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.true
        -- read status returned ready, so read
    -> ht_ioCmd.cmd00h -> ht_ioDataOut.false
    -> ht_ioCmd.cmd70h -> ht_ioDataOut.true
    -> READ_SPEC
assert READ_SPEC [F= READ_F_IMPL0
```

The complete list of tests undertaken are in the source file footer.csp available on the project website indicated in the acknowledgement section. The specifications and the process implementations have their basis in the ONFi document. All the tests performed on the failures model took 10 to 22 minutes to complete with exception of the first check which took 29 minutes. Some of the important tests performed on the model are listed below:

1. Deadlock and Livelock Freedom Checks using Failures and Failures Divergence Refinement respectively:

```
HOST_SW_TARGET_TWOLUNS :[deadlock free [F] ]
HOST_HW_TARGET_TWOLUNS :[deadlock free [F] ]
HOST_SW_TARGET_TWOLUNS :[livelock free [FD] ]
HOST_SW_TARGET_LUNHIDDEN :[livelock free [FD] ]
HOST_HW_TARGET_TWOLUNS :[livelock free [FD] ]
HOST_SW_ANYCMD_HIDDEN :[livelock free [FD] ]
HOST_SW_ANYCMD :[livelock free [FD] ]
```

2. Correctness Tests for *Read*, *PageParameter*, *BlockErase* and *MultiRead* Operation using Failures Refinement:

```
READ_SPEC [F= READ_F_IMPL0
READ_SPEC [F= READ_F_IMPL1
PP_SPEC [F= PP_F_IMPL0
PP_SPEC [F= PP_F_IMPL1
BE_SPEC [F= BE_F_IMPL0
BE_SPEC [F= BE_F_IMPL1
MULTIREAD_SPEC [F= MULTIREAD_F_IMPL0
MULTIREAD_SPEC [F= MULTIREAD_F_IMPL1
```

3. *Wrong* Implementations to ensure that the tests which should fail must fail:

```
BE_SPEC [F= BE_IMPL_F_WRONG0
BE_SPEC [F= BE_IMPL_F_WRONG1
```

Description of Model Setting	Time (min)	Nodes	Transitions
High Level Hiding + No Compression Applied	20.6	47787	117473
High Level Hiding + Normalise	18.5	11869	29452
High Level Hiding + ModelCompress	21.5	14696	39455
Low Level Hiding + Normalise	21.76	2393	5292
Low Level Hiding + ModelCompress	23.7	3827	9660

Table 4.3: Hiding and Compression Techniques Effect on State Space

```
READ_SPEC [F= READ_IMPL_F_WRONG1
MULTIREAD_SPEC [F= MULTIREAD_IMPL_F_WRONG0
```

4.5.5 "Deep Hiding" along with Model Compression Techniques available in FDR

While dealing with the state space problem, the FDR manual [For05] on Page 35 suggests: *"Hide all events at as low a level as is possible . . . any event that is to be hidden should be hidden the first time (in building up the process) that it no longer has to be synchronised at a higher level"*. The way the model was setup previously, was as follows:

```
LUN(lunID) = diamond(INITIAL_L_IDLE(lunID)
         [| li_events |] LI_IDLE(lunID))
TWOLUNS = LUN(lun0) ||| LUN(lun1)
TARGET = INITIAL_T_POWERON  [| tr_events |]
         READYBUSY(true,true)
TARGET_TWOLUNS = TARGET [| tl_events |] TWOLUNS
```

This clearly shows that the hiding of events was not applied at the first instant of the process setup. We changed the setup of the model as follows:

```
LUN(lunID) = diamond(INITIAL_L_IDLE(lunID)
        [| li_events |] LI_IDLE(lunID)) \ li_events
TWOLUNS = LUN(lun0) ||| LUN(lun1)
TARGET = (INITIAL_T_POWERON  [| tr_events |]
        READYBUSY(true,true)) \ tr_events
TARGET_TWOLUNS = (TARGET [| tl_events |]
        TWOLUNS) \ tl_events
```

Furthermore, by careful investigation we also found that there were two compression techniques i.e. `model_compress` and `normalise` which were neither applied automatically by FDR and nor by us. The remaining compression techniques i.e. `explicate`, `sbsim`, `tau_loop_factor` and `diamond` were already being used in the refinement step of states either by FDR or being manually applied. The details of these compression techniques i.e. how these techniques actually compress the model, are discussed in chapter 5 of [For05]. Table 4.3 lists the impact of these compressions and hiding of events on the state space. Here 'High Level Hiding' refers to the fact that hiding is applied at the top level while 'Low Level Hiding' refers to the hiding being as close to the point of definition of the relevant process as possible. We did these tests using FDR Explorer [FW09]. These tests were run on a multi-user timeshared machine, but one whose utilisation was pretty low. So, the timings mentioned here are just to indicate that these tests completed in a reasonable time limit.

The use of FDR Explorer was quite straightforward. For example, the commands used for testing `HOST_SW_TARGET_TWOLUNS` were:

```
$FDRHOME/bin/fdr2tix -insecure -nowindow
```

```
% source FDRExplorer.tcl
% inspectProcs ONFI-mandatory.csp
  HOST_SW_TARGET_TWOLUNS 0 0
```

After the application of hiding at low level, all the checks were again run to ensure the continued correctness of the model.

4.5.6 Tackling Full ONFi 2.1 Model

After having confidence that use of FDR Explorer and compression techniques could possibly be helpful in the compilation of the full model, hiding and compression were applied. But it again failed to compile. This is due to the fact that FDR is setup in such a way that it always performs complete ISM generation at the start, and it applies all compressions at a later stage. The failures we encountered occurred in the ISM generation phase. This fact came to our notice when FDR always reported 7023100 transitions (in the case of mandatory ONFI 2.1) in ISM generation in order to perform the first check and after that it started to compress the state-space during each of test runs. So, the application of hiding and compression techniques did not affect the performance.

After this failure, FDR support was contacted to get a 64-bit built of FDR so that we could possibly break the barrier of 32-bit limit for FDR paging. FDR support thankfully provided us with 64-bit built of the tool for Solaris machine. But even on 64-bit version of FDR, the ISM generation phase could not complete successfully, giving up after 6 hours of test running whereas the number of transitions at the point of failure was above 23 Million. The status of memory usage was investigated on the Solaris machine during the test, just before dying. The machine was consuming more than 30 GB of memory on the local disk as well as 10 GB on the physical memory. This clearly reflects that the full ONFi 2.1 model ISM is too large to handle in the present state.

4.6 Summary

We are now able to cover many of the operations in ONFi 2.1 model using full Failures-Divergence refinement checking, rather than just trace refinement. For ONFi 1.0, the total count of the *CSP* code was 1922 of which 1346 were automatically generated from the SCXML sources. Having upgraded to ONFi 2.1, the number of auto-generated lines of *CSP* has risen to 2070. Through the use of compression techniques available in the FDR toolkit and in particular by hiding the events deeper in the model, we were able to get compression of the state-space, i.e. low level hiding and `normalise` gave approximately 20 times more compression while in case of `model_compress`, this was a factor of about 12. However despite compression tricks and the use of FDR explorer, we still have not been able to compile the full ONFi model, which may represent the current limit of this model-checking technology. This is due to the fact that FDR does full compilation before any compressions are applied.

Chapter 5

Design

This chapter explains the translation process of a simple *Circus* example into its CSP_M equivalent. The translation process is explained through informal text, dividing the entire process into a number of steps. At first, the main action in the *Circus* example is composed containing only sequencing of the individual actions. Later on, the complexity of the construction of the main action is increased by adding external and internal choice operators. This inclusion of choices in the main action suggested that the translated description be in a normal form.

5.1 Translation Process for a Simple *Circus* Example

The entire process of translation is walked through by taking the simple *Circus* example shown in figure 5.1. A possible translation of this to *CSP* is shown in figure 5.2.

5.2 Translation Step-by-Step

We now show the steps required to get to this kind of end-result. In effect we make a series of passes that accumulate information about assignment variables and add these to parameter lists. Some of the intermediate steps are typically not well-formed, or even have different semantics to the original *Circus* process, but it is helpful to view the translation via these steps.

5.2.1 Step 1 – Scanning for the Variables in a *Circus* Process and Replacing Continuations for Skip

First we scan the process to identify all the variables, which in our example returns with the formal parameter fragment (x,y). Then we revise the definitions to add these parameters in. A particular trick we employ is to replace every *Skip* in the definition of N (say), by $CONT_N(x,y)$. These are placeholders to be further elaborated upon at a later stage.

$$A(x,y) \mathrel{\widehat=} x := 0 \mathbin{;} a \rightarrow CONT_A(x,y)$$
$$B(x,y) \mathrel{\widehat=} y := x+1 \mathbin{;} b.y \rightarrow CONT_B(x,y)$$
$$C(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow CONT_C(x,y)$$

In effect, each action is now parameterised with all the variables, and any final *Skip* are now replaced with a call to some as yet undetermined continuation process, with parameters being passed along.

$A \,{}_9^\circ\, B \,{}_9^\circ\, C$
where
$A \mathrel{\widehat=} x := 0 \,{}_9^\circ\, a \rightarrow Skip$
$B \mathrel{\widehat=} y := x + 1 \,{}_9^\circ\, b.y \rightarrow Skip$
$C \mathrel{\widehat=} c.(x + y) \rightarrow Skip$

Figure 5.1: A *Circus* Process example

$a \rightarrow b.1 \rightarrow c.1 \rightarrow Skip$

Figure 5.2: A possible translation

5.2.2 Step 2 – Removal of Assignment Commands and Replacing Expressions in the Parameter List

We then analyse the flow in each action and remove any assignments, noting and carrying along the changes made to the parameter list. The relevant form of the parameter list is then used to overwrite variables in expressions, and the parameter list of any $CONT_N$ encountered:

$A(x,y) \mathrel{\widehat=} a \rightarrow CONT_A(0,y)$
$B(x,y) \mathrel{\widehat=} b.(x+1) \rightarrow CONT_B(x,x+1)$
$C(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow CONT_C(x,y)$

5.2.3 Step 3 – Analysis of Main Action and Replacing Continuations with Corresponding Calling Action

Now we look at the main action, which is this case is

$A \,{}_9^\circ\, B \,{}_9^\circ\, C$

We note that here the continuation action for A is a call to B, for B is a call to C, and C has nothing following it, so its continuation is *Skip*. So we tailor our instances of A, B and C to fit the sequential flow of the main action, and now note that the initial A calls B when done, so the main action need only invoke A, with arbitrary starting parameter values.

$A(x,y)$
where
$A(x,y) \mathrel{\widehat=} a \rightarrow B(0,y)$
$B(x,y) \mathrel{\widehat=} b.(x+1) \rightarrow C(x,x+1)$
$C(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow Skip$

5.2.4 Step 4 – Inlining the Calling Actions and Propagating Parameter Changes

We can finish off if we so desire by "inlining" the calls and propagating the parameter changes along.

$$
\begin{array}{l}
A(x,y) \\
\textbf{where} \\
A(x,y) = a \rightarrow B(0,y) \\
B(0,y) = b.(0+1) \rightarrow C(0,0+1) \\
C(0,1) = c.(0+1) \rightarrow Skip
\end{array}
$$

This results in the translation shown in figure 5.2.

The advantage of this translation scheme is that the structure of the translation is a close match to that of the original, with assignments removed and parameters added. In particular, event prefixes are retained, with no events added or removed. However, as we look at more complicated cases, we see that the four-step translation process above is somewhat too simplistic.

5.2.5 Increasing Complexity

If we change the sequencing of the main action, then we instantiate the $CONT_N$ in a different way, so, for example with

$$
\begin{array}{l}
B \mathbin{;} C \mathbin{;} A \\
\textbf{where} \\
A \mathrel{\widehat{=}} x := 0 \mathbin{;} a \rightarrow Skip \\
B \mathrel{\widehat{=}} y := x+1 \mathbin{;} b.y \rightarrow Skip \\
C \mathrel{\widehat{=}} c.(x+y) \rightarrow Skip
\end{array}
$$

we obtain the following for Step 3:

$$
\begin{array}{l}
B(x,y) \\
\textbf{where} \\
A(x,y) \mathrel{\widehat{=}} a \rightarrow Skip \\
B(x,y) \mathrel{\widehat{=}} b.(x+1) \rightarrow C(x,x+1) \\
C(x,y) \mathrel{\widehat{=}} c.(x+y) \rightarrow A(x,y)
\end{array}
$$

reducing to

$$B(x,y)\,\textbf{where}\,B(x,y) \mathrel{\widehat{=}} b.(x+1) \rightarrow c.(x+x+1) \rightarrow a \rightarrow Skip$$

Things get more interesting if our main action is now a choice between the above two cases:

$$
\begin{array}{l}
(A \mathbin{;} B \mathbin{;} C) \Box (B \mathbin{;} C \mathbin{;} A) \\
\textbf{where} \\
A \mathrel{\widehat{=}} x := 0 \mathbin{;} a \rightarrow Skip \\
B \mathrel{\widehat{=}} y := x+1 \mathbin{;} b.y \rightarrow Skip \\
C \mathrel{\widehat{=}} c.(x+y) \rightarrow Skip
\end{array}
$$

After Step 2 we have:

$A(x,y) \mathrel{\widehat=} a \rightarrow CONT_A(0,y)$
$B(x,y) \mathrel{\widehat=} b.(x+1) \rightarrow CONT_B(x,x+1)$
$C(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow CONT_C(x,y)$

Now however, we find that each of A, B and C needs a different instantiation in each arm of the choice, so in effect we have to double them up:

$A_1(x,y) \Box B_2(x,y)$
where
$A_1(x,y) \mathrel{\widehat=} a \rightarrow B_1(0,y)$
$B_1(x,y) \mathrel{\widehat=} b.(x+1) \rightarrow C_1(x,x+1)$
$C_1(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow Skip$
$A_2(x,y) \mathrel{\widehat=} a \rightarrow Skip$
$B_2(x,y) \mathrel{\widehat=} b.(x+1) \rightarrow C_2(x,x+1)$
$C_2(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow A_2(x,y)$

It is quite clear that for more complex main actions, the number of instances will rise quite sharply.

This lead us to investigate the notion of a normal form that might simplify the translation, both conceptually, and in terms of size. To motivate this, consider the following example:

$(A \Box B) \mathbin{;} C$
where
$A \mathrel{\widehat=} x := 0 \mathbin{;} a \rightarrow Skip$
$B \mathrel{\widehat=} y := x+1 \mathbin{;} b.y \rightarrow Skip$
$C \mathrel{\widehat=} c.(x+y) \rightarrow Skip$

We might try doing Step 1, then setting A and B to have C as a continuation, so obtaining:

$(A \Box B) \mathbin{;} C$
where
$A(x,y) \mathrel{\widehat=} a \rightarrow C(0,y)$
$B(x,y) \mathrel{\widehat=} y := x+1 \mathbin{;} b.(x+1) \rightarrow C(x,x+1)$
$C(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow Skip$

We then replace the main action with $A(x,y) \Box B(x,y)$, noting that these two now call C when they are done. In effect we are using the CSP law $(P \Box Q) \mathbin{;} R = (P \mathbin{;} R) \Box (Q \mathbin{;} R)$. We obtain:

$A(x,y) \Box B(x,y)$
where
$A(x,y) \mathrel{\widehat=} a \rightarrow C(0,y)$
$B(x,y) \mathrel{\widehat=} b.(x+1) \rightarrow C(x,x+1)$
$C(x,y) \mathrel{\widehat=} c.(x+y) \rightarrow Skip$

which reduces to:

$$D(x,y)\ \textbf{where}\ D(x,y) \mathrel{\widehat=} (a \to c.y \to Skip) \Box (b.(x+1) \to c.(2x+1) \to Skip)$$

This seemed to suggest a normal form that brought choices out in front of sequential composition — something like

$$\bigsqcap(\Box \text{sequential forms})$$

where step laws are used to factor out parallelism. Here "sequential forms" are actions built out of prefix, assignment, sequential composition and action calls only — no choice or parallel operators, and their associated branching.

5.3 Summary

In this chapter, we explain the translation process of a simple *Circus* example into its CSP_M equivalent. The translation process is explained through informal text, dividing the entire process into a number of steps. The first step is to search for the variables in a *Circus* program. The second step is the removal of the assignment commands and substitution of the expressions into the parameter list. Then, the next step involves analysis of the main action and replacing continuations with the corresponding calling action. The final step is to do inlining of the calling actions and propagating the parameter changes. At first, the main action in the *Circus* example is composed containing only sequencing of the individual actions. Later on, the complexity of the construction of the main action is increased by adding external and internal choice operators. This inclusion of choices in the main action suggested the translated description using a normal form. The next chapter is the most important one as it desribes the semantic justification of the link between *Circus* and CSP_M through the mathematical proofs.

Chapter 6

Semantics

This chapter describes the semantics of the target languages, i.e. Simple*CSP* and Simple*Circus*. First, the UTP semantics of the languages are formulated, having its basis in [OCW09, CW04]. The meaning of language constructs considered in Simple*CSP* and Simple*Circus* are explained through informal text. Following is a description of approach adopted for establishing the link between the languages. The chapter concludes by including the proofs of the theorems and lemmas suggested for the link.

6.1 The UTP Semantics of Simple*CSP* and Simple*Circus*

6.1.1 Semantics of Simple*CSP*

The observation variables were briefly introduced on page 13, section 2.4. The variables ok, ok' depict the successful initiation and termination of the program. So, these are of boolean type, showing either a true or false condition. The variables $wait, wait'$ depict the waiting status of a program for an interaction with its environment. So, these are of type boolean, showing the waiting status as either true or false. The variables tr, tr' are for the initial and final traces of a program. These traces record event histories as a sequence of events. The variables ref, ref' result in a set, capturing the events being refused by a program during execution, given that the events recorded in tr, tr' have already occurred.

Observations for Simple*CSP*:

$$\begin{aligned} ok, ok' &: \mathbb{B} \\ wait, wait' &: \mathbb{B} \\ tr, tr' &: \Sigma^* \\ ref, ref' &: \mathscr{P}\,\Sigma \end{aligned}$$

Healthiness conditions of *CSP*, with their informal meaning, were briefly introduced on page 14, section 2.4.2.

Healthiness for Simple*CSP*:

$$\begin{aligned} \mathbf{R1}(P) &\mathrel{\widehat=} P \land tr \le tr' \\ \mathbf{R2}(P) &\mathrel{\widehat=} \exists s \bullet P[s, s ^\frown (tr' - tr)/tr, tr'] \\ \mathbf{R3}(P) &\mathrel{\widehat=} I\!I \lhd wait \rhd P \\ I\!I &\mathrel{\widehat=} DIV \lor ok' \land wait' = wait \land tr' = tr \land ref' = ref \\ DIV &\mathrel{\widehat=} \neg ok \land tr \le tr' \end{aligned}$$

$$
\begin{array}{rcl}
\mathbf{R} & \widehat{=} & \mathbf{R3} \circ \mathbf{R2} \circ \mathbf{R1} \\
\mathbf{CSP1}(P) & \widehat{=} & P \lor DIV \\
\mathbf{CSP2}(P) & \widehat{=} & P \mathbin{;} J \\
\mathbf{CSP4}(P) & \widehat{=} & P;\ SKIP \\
J & \widehat{=} & (ok \Rightarrow ok') \land wait' = wait \land tr' = tr \land ref' = ref \\
\mathbf{CSP} & \widehat{=} & \mathbf{CSP1} \circ \mathbf{CSP2} \circ \mathbf{CSP4} \\
P \vdash Q & \widehat{=} & ok \land P \Rightarrow ok' \land Q, \quad ok, ok' \notin P, Q
\end{array}
$$

Having expansions of **R** are useful. Given that **R1** and $I\!I$ are **R2** healthy, and $I\!I$ is **R1**-healthy the most useful expansion is:

$$
\begin{array}{rl}
& \mathbf{R}(P) \\
= & \mathbf{R3}(\mathbf{R1}(\mathbf{R2}(P))) \\
= & I\!I \lhd wait \rhd (\exists s \bullet P[s, s \frown (tr' - tr)/tr, tr']) \land tr \le tr'
\end{array}
$$

In most case, our language definition fragments are **R2**-healthy so we often ignore it quite safely, to get the following "standard expansion" of $\mathbf{R}(P)$:

$$
\mathbf{R}(P) \quad = \quad I\!I \lhd wait \rhd P \land tr \le tr'
$$

6.1.2 Exploring the Semantics of Parameters

The key idea is that we have named, parameterised action definitions of the form:

$$
N(x_1, x_2, \ldots, x_n) \quad \widehat{=} \quad A
$$

Here the x_i are parameters that denote expressions.

We can view the meaning of this as:

$$
\lambda\, x_1, x_2, \ldots, x_n \bullet A
$$

Then the meaning of an invocation of N

$$
N(e_1, e_2, \ldots, e_n)
$$

is

$$
A[e_1, e_2, \ldots, e_n / x_1, x_2, \ldots, x_n]
$$

When recursion is involved, things get a little more complicated:

$$
R(x_1, x_2, \ldots, x_n) \quad \widehat{=} \quad F(R)
$$

We obtain the following fixpoint equation:

$$
R = \mu\, X \bullet \lambda\, x_1, \ldots, x_n \bullet F(X)
$$

Regardless of which fixed point we use, the following is a fold/unfold law:

$$R = F(R)$$

If recursion is guarded, then we should also have a unique fixed-point principle, so:

$$guarded(F) \wedge R = F(R) \wedge S = F(S) \Rightarrow R = S$$

Now we introduce the formal semantics for Simple*CSP*.

$$\begin{array}{rcl} N(x_1,\ldots,x_n) \mathrel{\widehat=} A & \equiv & N = \mu X \bullet \lambda\, x_1,\ldots,x_n \bullet A[X/N] \\ N & = & A[\lambda\, x_1,\ldots,x_n \bullet A/N] \end{array}$$

The deadlock or *Stop* cannot engage in any event and is always waiting. So, here the final trace tr' will be equal to the initial trace tr and the variable $wait'$ will be true. *Stop* must refuse all the events as it represents a deadlock. The definition of *Stop* allows any possible refusal set, including the one that contains all events, by not mentioning ref or ref' in the definition.

$$Stop \mathrel{\widehat=} \mathbf{R}(true \vdash wait' \wedge tr' = tr)$$

Skip is an action which terminates immediately. Here the final trace tr' is equal to the initial trace tr. The final waiting status $wait'$ will be false.

$$\begin{array}{rcl} Skip & \mathrel{\widehat=} & \mathbf{R}(\exists\, ref \bullet II) \\ & \equiv & \mathbf{R}(true \vdash \neg wait' \wedge tr' = tr) \end{array}$$

The *CSP* prefixing action is **CSP1** healthy. It never diverges. On termination, it establishes the result of do_A, defined below. In do_A, while waiting $(wait' = true)$, it simply requires that a is not being refused, while, once waiting is over $(wait' = false)$, then the trace has been extended with a.

$$\begin{array}{rcl} a \rightarrow Skip & \mathrel{\widehat=} & \mathbf{CSP1}(ok' \wedge do_A(a)) \\ do_A(a) & \mathrel{\widehat=} & \Phi(a \notin ref' \lhd wait' \rhd tr' = tr ^\frown \langle a \rangle) \\ \Phi(A) & \mathrel{\widehat=} & \mathbf{R}(A) \wedge B = \mathbf{R}(A \wedge B) \\ B & \mathrel{\widehat=} & tr' = tr \lhd wait' \rhd tr < tr' \end{array}$$

The definition of prefix, in which after an event a, the action A is invoked, can be split into a sequential composition of $a \rightarrow Skip$ followed by the action A. Similar definition can be devised for input and output prefixing action. An event can be defined as a pair (c, e) where c is the channel and e is the value to be communicated on the channel.

$$\begin{array}{rcl} a \rightarrow A & \mathrel{\widehat=} & a \rightarrow Skip \mathbin{;} A \\ c!e \rightarrow A & \mathrel{\widehat=} & c.e \rightarrow Skip \mathbin{;} A \\ c?v \rightarrow A & \mathrel{\widehat=} & \Box_k \bullet c.k \rightarrow A[k/v], \quad k \in type(v) \end{array}$$

The sequential composition is defined as a relational sequence, instead of being defined as reactive design. Here, the relational sequence is defined as an existential quantifier over the intermediate state of the UTP

observation variables i.e. $ok, wait, tr, ref$.

$$A \fatsemi B \;\widehat{=}\; \exists Obs_m \bullet A[Obs_m/Obs'] \wedge B[Obs_m/Obs] \quad \textbf{where}\, Obs = \{ok, wait, tr, ref\}$$

Internal choice operator is defined as a disjunction between the actions A and B, showing that either of the actions can proceed.

$$A \sqcap B \;\widehat{=}\; A \vee B$$

The external choice is **CSP2** healthy. If a deadlock ($Stop$) occurs, it is a conjuction of actions A and B. If deadlock does not occur, it is a disjunction of actions A and B. $Stop$ being true means we are healthy but waiting. During this time, we want A and B to agree on refusals (see definition of $do_A(a)$ above), when $wait'$ is true. When $Stop$ no longer holds, $wait'$ is false and so we want the behaviour of whichever of A and B can do the event offered by the environment.

$$A \,\Box\, B \;\widehat{=}\; \mathbf{CSP2}((A \wedge B) \lhd Stop \rhd (A \vee B))$$

For hiding operator, if action A reaches such a stable state, in which it cannot perform any further events in H, then the action $A \setminus H$ has also reached such state. Here, $tr_H - tr$ and $tr' - tr$ depict the new events of action A during hiding. $Skip$ is to ensure the inclusion of possible divergences by hiding actions.

$$A \setminus H \;\widehat{=}\; \mathbf{R}(\exists tr_H \bullet A[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H) \fatsemi Skip$$

The parallel construct is defined as existensial quantifier of observation variables of actions A and B over the conjuction of the actions. Here, both A and B run together, but we rename their after-observations so we can distinguish their outcomes. We then merge these outcomes to give the overall parallel outcome.

$$A \parallel_S B \;\widehat{=}\; \mathbf{R}\left(\begin{array}{l} \exists Obs_A, Obs_B \bullet \\ \quad A[Obs_A/Obs'] \wedge B[Obs_B/Obs'] \\ \wedge\, ok' = ok_A \wedge ok_B \\ \wedge\, wait' = wait_A \vee wait_B \\ \wedge\, ref' \subseteq (ref_A \cup ref_B) \cap S \cup (ref_A \cap ref_B) \setminus S \\ \wedge\, tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \end{array}\right) \fatsemi Skip$$

In a guarded action, if the condition given in expression e is true, the action A will take place, otherwise, it will deadlock.

$$e \& A \;\widehat{=}\; A \lhd e \rhd Stop$$

The parameterised call to an action is through calling the action and replacing the corresponding expressions into the parameter list.

$$N(e_1, \ldots, e_n) \;\widehat{=}\; B[e_1, \ldots, e_n/x_1, \ldots, x_n] \quad \textbf{given}\, N(x_1, \ldots, x_n) \;\widehat{=}\; B$$

In above, two auxiliary functions are used:

$$\begin{array}{rl} t \upharpoonright S & \text{trace } t \text{ with all elements in } S \text{ removed} \\ t_1 \parallel_S t_2 & \text{set of all interleavings of } t_1, t_2 \text{ that synchronise on } S \end{array}$$

6.1.3 Semantics of Simple*Circus*

The main difference between observation variables of Simple*Circus* and Simple*CSP* is the inclusion of additional $state, state'$ which depict the status of the state variables in the program. Here, it is defined as a partial function from a variable to its value.

Observations for Simple*Circus*:

$$\begin{array}{rcl} ok, ok' & : & \mathbb{B} \\ wait, wait' & : & \mathbb{B} \\ tr, tr' & : & \Sigma^* \\ ref, ref' & : & \mathscr{P}\Sigma \\ state, state' & : & Var \rightarrow Value \end{array}$$

In the definition of healthiness conditions of Simple*Circus*, **R** and **CSP** are renamed to **S** and **CXS**, to avoid confusion.

Now we give the definitions of healthiness conditions of Simple*Circus*.

$$\begin{array}{rcl} \mathbf{S1}(P) & \widehat{=} & P \wedge tr \leq tr' \\ \mathbf{S2}(P) & \widehat{=} & \exists s \bullet P[s, s \frown (tr' - tr)/tr, tr'] \\ \mathbf{S3}(P) & \widehat{=} & (\exists state' \bullet II) \lhd wait \rhd P \\ II & \widehat{=} & DIV \vee ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge state' = state \\ DIV & \widehat{=} & \neg ok \wedge tr \leq tr' \\ \mathbf{S} & \widehat{=} & \mathbf{S3} \circ \mathbf{S2} \circ \mathbf{S1} \\ \mathbf{CXS1}(P) & \widehat{=} & P \vee DIV \\ \mathbf{CXS2}(P) & \widehat{=} & P \mathbin{;} JX \\ \mathbf{CXS4}(P) & \widehat{=} & P \mathbin{;} Skip \\ JX & \widehat{=} & (ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge state' = state \\ P \vdash Q & \widehat{=} & ok \wedge P \Rightarrow ok' \wedge Q, \quad ok, ok' \notin P, Q \end{array}$$

Here, JX definition includes $state, state'$ as well.

The "standard expansion" of $\mathbf{S}(P)$:

$$\mathbf{S}(P) \quad = \quad (\exists state' \bullet II) \lhd wait \rhd P \wedge tr \leq tr'$$

An "alternate expansion" of $\mathbf{S}(P)$:

$$\mathbf{S}(P) \quad = \quad (\exists state' \bullet II) \lhd wait \rhd (P \wedge tr \leq tr')$$

The work in [BGW09] explains the motivation behind **S3** (**R3** with state-hiding). [BGW09] investigated the interactions between program variable state visibility and communication behaviour in state-rich CSP-like processes, using the UTP framework. This gave the result that, if the variable state is visible during the wait

state of communication, ordinary programs put together in a normal way ended up with miraculous behaviour — to be avoided in a theory of feasible programming.

Now the formal semantics for Simple*Circus* is given. Here, many of the definition of the language constructs involve $state, state'$ observation variables, which were not there in the case of Simple*CSP*. Most of the definitions are essentially the same as those for Simple*CSP*, and we only really need to consider the assignment command, and the parallel construct.

$$
\begin{array}{rcl}
N(x_1,\ldots,x_n) \mathrel{\widehat=} A & \equiv & N = \mu X \bullet \lambda x_1,\ldots,x_n \bullet A[X/N] \\
N & = & A[\lambda x_1,\ldots,x_n \bullet A/N]
\end{array}
$$

$$
\begin{array}{rcl}
Stop & \mathrel{\widehat=} & \mathbf{S}(true \vdash wait' \wedge tr' = tr) \\
Skip & \mathrel{\widehat=} & \mathbf{S}(true \vdash \neg wait' \wedge tr' = tr \wedge state' = state) \\
A \mathbin{;} B & \mathrel{\widehat=} & \exists Obs_m \bullet A[Obs_m/Obs'] \wedge B[Obs_m/Obs] \\
 & & \mathbf{where}\, Obs = \{ok, wait, tr, ref, state\} \\
c \rightarrow Skip & \mathrel{\widehat=} & \mathbf{S}\left(true \vdash \left(\begin{array}{l} tr' = tr \wedge c \notin ref' \\ \lhd wait' \rhd \\ tr' = tr \frown \langle c \rangle \wedge state' = state \end{array}\right)\right) \\
c \rightarrow A & \mathrel{\widehat=} & (c \rightarrow Skip) \mathbin{;} A \\
c!e \rightarrow A & \mathrel{\widehat=} & (c.e \rightarrow Skip) \mathbin{;} A \\
c?x \rightarrow A & \mathrel{\widehat=} & \Box_{v \in typeof(x)}\, c.v \rightarrow x := v \mathbin{;} A \\
A \sqcap B & \mathrel{\widehat=} & A \vee B \\
A \Box B & \mathrel{\widehat=} & \mathbf{CXS2}((A \wedge B) \lhd Stop \rhd (A \vee B)) \\
A \setminus H & \mathrel{\widehat=} & \mathbf{S}(\exists tr_H \bullet A[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \downharpoonright H) \mathbin{;} Skip \\
N(e_1,\ldots,e_n) & \mathrel{\widehat=} & B[e_1,\ldots,e_n/x_1,\ldots,x_n] \\
 & & \mathbf{given}\, N(x_1,\ldots,x_n) \mathrel{\widehat=} B \\
e \& A & \mathrel{\widehat=} & A \lhd e \rhd Stop
\end{array}
$$

In the Simple*Circus* model of parallelism, both sides run on their own copy of the initial state, and the two final states are merged at the end. A requirement is that the set of variables written by each side are disjoint from each other. Also note that changes to variables done by one side are not visible to the other side.

$$
\begin{array}{rcl}
A[_U|_S|_V]B & \mathrel{\widehat=} & \exists Obs_A, Obs_B \bullet \\
 & & \quad A[Obs_A/Obs'] \wedge B[Obs_B/Obs'] \\
 & & \wedge\, ok' = ok_A \wedge ok_B \\
 & & \wedge\, wait' = wait_A \vee wait_B \\
 & & \wedge\, tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \\
 & & \wedge \left(\begin{array}{c} ref' \subseteq (ref_A \cup ref_B) \cap S \cup (ref_A \cap ref_B) \setminus S \\ \lhd wait' \rhd \\ state' = state \setminus (U \cup V) \oplus state_A \upharpoonright U \oplus state_B \upharpoonright V \end{array}\right)
\end{array}
$$

The assignment command was not included in Simple*CSP*. In Simple*CSP*, these assignment commands turn into the parameterised call to an action, where the corresponding expressions are substituted in the parameter list. The UTP definition of the assignment command in Simple*Circus* is defined as a reactive design. Here, the final status of state variables i.e. $state'$ is equal to the sum of unchanged state variables and the changed one having a mapping from x to expression e, evaluated in the before-state.

$$x := e \quad \widehat{=} \quad \mathbf{S}(true \vdash \neg wait' \wedge tr' = tr \wedge state' = state \oplus \{x \mapsto e_{state}\})$$

We use the following auxiliary functions that act on states:

$state \setminus V$	state with all variables in V removed
$state \upharpoonright V$	state with only variables in V retained

6.2 The Link between Simple*Circus* and Simple*CSP*

Here, we make a distinction between the notation of constructs of Simple*Circus* and Simple*CSP*. The subscript C is used to distinguish the *CSP* version of a construct e.g. ($Stop_C$) from the *Circus* version of the construct i.e. ($Stop_X$). Here, X is the subscript for the *Circus* version of the construct.

The observation variables defined for Simple*Circus* and Simple*CSP* have as a difference the presence of the variables $state_X$ and $state'_X$ in Simple*Circus*. So, the link between Simple*Circus* and Simple*CSP* has to deal with this difference. The need here is a linking predicate that connects these:

$$L^C_X \quad \widehat{=} \quad ok_C = ok_X \wedge ok'_C = ok'_X \wedge ...ref'_C = ref'_X$$
$$\wedge \text{ relationship of } state_X \text{ and } state'_X$$

Here, we do some pilot proofs of concrete translations, to explore the usefulness of the semantics and to get some idea of useful laws that might help simplify proofs.

First we introduce some notation:

P_X	A Simple*Circus* program
$\mathscr{T}$	The translation predicate transformer
P_C	A Simple*CSP* program

If P_C is the translation of P_X, then we want to prove that they have the same behaviour, when we ignore state. So, given

$$P_C = \mathscr{T} P_X$$

we want to show that

$$P_C = (\exists state, state' \bullet P_X)$$

But after mathematical investigation, it turned out that the best we can get is a refinement relation, so that

$$P_C \sqsubseteq (\exists state, state' \bullet P_X)$$

This effectively defines the connecting link between *Circus* and *CSP* UTP theories.

So, the big theorem for the translation between Simple*Circus* and Simple*CSP* is

Conjecture 6.2.1 *For all Simple*Circus* programs P_X:*

$$\mathscr{T}(P_X) \sqsubseteq (\exists state, state' \bullet P_X)$$

We want to show that the CSP_M description produced by the translation covers the behaviour of the original *Circus* process with the state hidden. It would be very nice if simply hiding the state was enough, but we find

the issue is more complex.

Here are some possible “linking pairs” in Simple*Circus* and Simple*CSP* worlds:

P_C	*Rel*	P_X	*Theorem*	Proof?	By
$Stop_C$	$=$	$\exists state, state' \bullet Stop_X$	B.2.1	Pg. 123	Dr. Butterfield
$Skip_C$	$=$	$\exists state, state' \bullet Skip_X$	6.2.1	Pg. 49	–
$a \rightarrow_C Skip_C$	$=$	$\exists state, state' \bullet a \rightarrow_X Skip_X$	B.1.1	Pg. 119	Dr. Butterfield
$(A_C \mathbin{;}_C B_C)$	$\sqsubseteq$	$(\exists state, state' \bullet A_X \mathbin{;}_X B_X)$	6.2.2	Pg. 50	–
$A_C \sqcap B_C$	$\sqsubseteq$	$(\exists state, state' \bullet A_X \sqcap B_X)$	6.2.3	Pg. 50	–
$A_C \mathbin{;}_C J_C$	$\sqsubseteq$	$(\exists state, state' \bullet A_X \mathbin{;}_X J_X)$	B.2.5	Pg. 126	Dr. Butterfield
$A_C \Box B_C$	$\sqsubseteq$	$(\exists state, state' \bullet A_X \Box B_X)$	6.2.4	Pg. 51	–
$A_C \parallel B_C$	$\sqsubseteq$	$(\exists state, state' \bullet A_X \, {}_S\|_T \, B_X)$	6.2.5	Pg. 52	–
$a \rightarrow A_C$	$\sqsubseteq$	$(\exists state, state' \bullet a \rightarrow A_X)$	6.2.6	Pg. 53	–
$A_C \setminus_C H$	$\sqsubseteq$	$(\exists state, state' \bullet A_X \setminus_X H)$	6.2.7	Pg. 54	–
$\mathbf{Rn}(A_C)$	$\sqsubseteq$	$\exists state, state' \bullet \mathbf{Sn}(A_X), \quad \mathbf{n} \in \mathbf{1} \ldots \mathbf{3}$	B.2.6	Pg. 128	–
$\mathbf{CSPn}(A_C)$	$\sqsubseteq$	$\exists state, state' \bullet \mathbf{CXSn}(A_X), \quad \mathbf{n} \in \mathbf{1} \ldots \mathbf{5}$	B.2.7	Pg. 129	–

Note: In the mathematical proofs, the sequential composition case was not trivial. Here, the left and right hand side were shown to be non-equivalent. Instead, the relationship between $;_X$ and $;_C$ is proved to be a refinement, not an equality. Because we obtained a refinement for the $;$ operator, all other results on general predicates A, B are forced to be refinements as well, by the monotonicity of the operators.

6.2.1 Assignment Command Handling

In the link pairs table above, the assignment command is missing. Before going to start the mathematical proofs, it is important to discuss the handling of the assignment command in the translation. We have already discussed the semantics of parameters in Section 6.1.2.

The semantics of assignment, in both *Circus*, and in regular imperative languages, has the effect, when followed sequentially by some action, of acting like a substitution, i.e. the following rule holds in the case of the assignment command:

$$x := e;\ A = A[e/x]$$

Here, note that if we have parameters with an action, e.g. $A(x,y)$, then the substitution $A(x,y)[e/x]$ can immediately be written as $A(e,y)$. This is the intuition behind how we translate assignments.

From the translation perspective, the above rule is ensured through a seperate formalised step in the translation process.

In Chapter 7, we formalise the translation steps. The steps are implemented as functions. In the process, the first step is “Ensure Assignment Continuation”. If an assignment appears without a following sequential composition,, it appends a sequentially composed *Skip* in front of it i.e. $x := e$ becomes $x := e$; *Skip*. This is to ensure that the assignment command is always followed by something. There is a following step in the translation which is named as “Add Continuation Call”. It replaces each *Skip* with a continuation marker followed by the action name. Substitution of expression e is done in the parameter list attached to the action call. Later on, this continuation marker along with the action name is replaced by the appropriate action call, determined by the analysis of the main action.

6.2.2 Mathematical Proofs

Here, for the proofs, the assumptions throughout are:

$$\begin{aligned} A_C &\sqsubseteq \exists state, state' \bullet A_X \\ B_C &\sqsubseteq \exists state, state' \bullet B_X \end{aligned}$$

Theorem 6.2.1

$$Skip_C = (\exists state, state' \bullet Skip_X)$$

Proof: Here, we adopt the strategy of reducing the left and right hand side of the equation to the same form.

$$LHS = Skip_C$$

Now, according to the definition of the *Skip* construct in the *CSP* world, the left hand side will be equal to:

$$= II \lhd wait \rhd (true \vdash \neg wait' \wedge tr' = tr) \wedge tr \leq tr'$$

Now, considering the right hand side:

$$RHS = (\exists state, state' \bullet Skip_X)$$

According to the definition of the *Skip* construct in the *Circus* world, the equation will be:

$$= (\exists state, state' \bullet \mathbf{S}(true \vdash \neg wait' \wedge tr' = tr \wedge state' = state))$$

Expanding the above equation by applying the definition of the healthiness condition **S**, will make the above equation as:

$$= (\exists state, state' \bullet (\exists state \bullet II_X)) \lhd wait \rhd (true \vdash \neg wait' \wedge tr' = tr \wedge state' = state) \wedge tr \leq tr'$$

Distributing the quantifier $\exists$:

$$\begin{aligned} = (\exists state, state' \bullet (\exists state \bullet II_X)) \lhd wait \rhd (true \vdash \neg wait' \wedge tr' = tr \wedge \\ (\exists state, state' \bullet state' = state)) \wedge tr \leq tr' \end{aligned}$$

Now, we apply the definition of Lemma B.2.2, available on page 124.

$$= II_X \lhd wait \rhd (true \vdash \neg wait' \wedge tr' = tr \wedge (\exists state, state' \bullet state' = state)) \wedge tr \leq tr'$$

Applying the one point rule will reduce the above equation to the following:

$$= II_X \lhd wait \rhd (true \vdash \neg wait' \wedge tr' = tr \wedge True) \wedge tr \leq tr'$$

Simplifying above equation:

$$= II_X \lhd wait \rhd (true \vdash \neg wait' \wedge tr' = tr) \wedge tr \leq tr'$$

This form of the equation makes the proof complete, as the left hand side also has the same form.

Theorem 6.2.2

$$(A_C \mathbin{;}_C B_C) \sqsubseteq (\exists state, state' \bullet A_X \mathbin{;}_X B_X)$$

Proof: Here, we show the left hand side is *refined by* the right hand side. This theorem is unique from others. Here, the left and right hand side were originally hoped to be equivalent. Instead, the relationship between $;_X$ and $;_C$ is emerged as a refinement, not an equality.

$$L.H.S = (A_C \mathbin{;}_C B_C)$$

According to the definition of A_C and B_C:

$$= (\exists state, state' \bullet A_X) \mathbin{;}_C (\exists state, state' \bullet B_X)$$

This prove completes by the application of theorem B.2.4, on page 125.

$$\sqsubseteq \qquad \exists state, state' \bullet A_X \mathbin{;}_X B_X$$

Theorem 6.2.3

$$A_C \sqcap B_C \sqsubseteq (\exists state, state' \bullet A_X \sqcap B_X)$$

Proof: Here, we adopt the strategy of reducing the left and right hand side of the equation to the same form.

$$L.H.S = A_C \sqcap B_C$$

According to the definition of internal choice in *CSP* world, we get,

$$= A_C \vee B_C$$

Expanding the definitions of A_C and B_C:

$$\sqsubseteq (\exists state, state' \bullet A_X) \vee (\exists state, state' \bullet B_X)$$

The distributive property of $\exists$ simplifies the above expression to the following:

$$= \exists state, state' \bullet (A_X \vee B_X)$$

Now, we can apply the definition of internal choice construct in *Circus* world.

$$= (\exists state, state' \bullet A_X \sqcap B_X)$$

Hence, the required form is acquired.

Theorem 6.2.4

$$A_C \Box B_C \sqsubseteq (\exists state, state' \bullet A_X \Box B_X)$$

Proof: Here, we adopt the strategy of reducing the left and right hand side of the equation to the same form. We start the proof by taking the right hand side of the theorem.

$$R.H.S = (\exists state, state' \bullet A_X \Box B_X)$$

According to the definition of external choice in the *Circus* world, we get:

$$= (\exists state, state' \bullet \mathbf{CXS2}((A_X \wedge B_X) \lhd Stop_X \rhd (A_X \vee B_X)))$$

Expanding the above equation by applying the definition of the healthiness condition **CXS2**, will make the above equation become:

$$= (\exists state, state' \bullet [((A_X \wedge B_X) \lhd Stop_X \rhd (A_X \vee B_X)) \mathbin{;} JX])$$

Distributing the predicate $\exists state, state'$, we get:

$$= \exists state, state' \bullet ((A_X \wedge B_X) \lhd Stop_X \rhd (A_X \vee B_X)) \mathbin{;} \exists state, state' \bullet JX$$

Expanding the above equation by applying the definition of J in *Circus* world will make the above equation be:

$$\begin{aligned} = \exists state, state' \bullet ((A_X \wedge B_X) \lhd Stop_X \rhd (A_X \vee B_X)) \mathbin{;} \exists state, state' \bullet (ok \Rightarrow ok') \\ \wedge\ wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge state' = state \end{aligned}$$

Applying the one point rule will reduce the above equation to the following:

$$= \exists state, state' \bullet ((A_X \wedge B_X) \lhd Stop_X \rhd (A_X \vee B_X)) \mathbin{;} (ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref$$

Applying the definition of J in *CSP* world will reduce the equation as follows:

$$= \exists state, state' \bullet ((A_X \wedge B_X) \lhd Stop_X \rhd (A_X \vee B_X)) \mathbin{;} J_C$$

Now, applying the definition of the $\lhd\rhd$ construct:

$$= \exists state, state' \bullet (Stop_X \wedge (A_X \wedge B_X) \vee \neg Stop_X \wedge (A_X \vee B_X)) \mathbin{;} J_C$$

The distributive property of $\exists state, state'$ will make the equation become:

$$\begin{aligned} = ((\exists state, state' \bullet Stop_X \wedge (\exists state, state' \bullet A_X \wedge \exists state, state' \bullet B_X)) \\ \vee (\exists state, state' \bullet \neg Stop_X \wedge (\exists state, state' \bullet A_X \vee \exists state, state' \bullet B_X))) \mathbin{;} J_C \end{aligned}$$

Now we apply theorem B.2.1, on page 123.

$$= ((Stop_C \wedge (\exists state, state' \bullet A_X \wedge \exists state, state' \bullet B_X)) \vee (\neg Stop_C \wedge (\exists state, state' \bullet A_X \vee \exists state, state' \bullet B_X))) \mathbin{;} J_C$$

Reducing above equation by using definitions of $\lhd\rhd$, A_C and B_C:

$$\sqsubseteq ((A_C \wedge B_C) \lhd Stop_C \rhd (A_C \vee B_C)) \mathbin{;} J_C$$

Using the definition of healthiness condition **CSP2**, we get:

$$= \mathbf{CSP2}((A_C \wedge B_C) \lhd Stop_C \rhd (A_C \vee B_C))$$

Now, we can apply the definition of external choice construct in *CSP* world.

$$= A_C \,\square\, B_C$$

This completes our proof as we have matched the expression of the left hand side of the theorem.

Theorem 6.2.5

$$A_C \parallel_S B_C \sqsubseteq (\exists state, state' \bullet A_X[_U|_S|_V]B_X)$$

Proof: Here, we adopt the strategy of reducing the left and right hand side of the equation to the same form. We start the proof by taking the left hand side of the theorem.

$$L.H.S = A_C \parallel_S B_C$$

Applying the definition of parallel construct in the *CSP* world, we get the following expression.

$$= \left(\begin{array}{l} \exists Obs_A^C, Obs_B^C \bullet \\ \quad A_C[Obs_A^C/Obs'] \wedge B_C[Obs_B^C/Obs'] \\ \quad \wedge\, ok' = ok_A^C \wedge ok_B^C \\ \quad \wedge\, wait' = wait_A^C \vee wait_B^C \\ \quad \wedge\, ref' \subseteq (ref_A^C \cup ref_B^C) \cap S \cup (ref_A^C \cap ref_B^C) \setminus S \\ \quad \wedge\, tr' - tr \in (tr_A^C - tr) \parallel_S (tr_B^C - tr) \end{array} \right) \,;_C Skip_C$$

Now, taking right hand side of the theorem and solving it to get the expanded form of left hand side.

$$R.H.S = (\exists state, state' \bullet A_X[_U|_S|_V]B_X)$$

According to the definition of parallel construct in *Circus* world, we get:

$$= \exists state, state' \bullet \left(\begin{array}{l} \exists Obs_A^X, Obs_B^X \bullet \\ \quad A_X[Obs_A^X/Obs'] \wedge B_X[Obs_B^X/Obs'] \\ \quad \wedge\, ok' = ok_A^X \wedge ok_B^X \\ \quad \wedge\, wait' = wait_A^X \vee wait_B^X \\ \quad \wedge\, tr' - tr \in (tr_A^X - tr) \parallel_S (tr_B^X - tr) \\ \quad \wedge \left(\begin{array}{c} ref' \subseteq (ref_A^X \cup ref_B^X) \cap S \cup (ref_A^X \cap ref_B^X) \setminus S \\ \lhd wait' \rhd \\ state' = state \setminus (U \cup V) \oplus state_A^X \upharpoonright U \oplus state_B^X \upharpoonright V \end{array} \right) \end{array} \right)$$

In the Simple*Circus* model of parallelism, both sides run on their own copy of the initial state, and the two final states are merged at the end. A requirement is that the set of variables written by each side are disjoint from each other. Furthermore, the changes to variables done by one side are not visible to the other side. Therefore, we can partially apply the one-point rule, as there are no free occurrences of state', which means the statement $state' = \ldots$ reduces to true, but we keep the now-vacuous quantification of state' because it makes later steps

simpler.

$$= \exists state, state' \bullet \left(\begin{array}{l} \exists Obs_A^X, Obs_B^X \bullet \\ \quad A_X[Obs_A^X/Obs'] \wedge B_X[Obs_B^X/Obs'] \\ \quad \wedge ok' = ok_A^X \wedge ok_B^X \\ \quad \wedge wait' = wait_A^X \vee wait_B^X \\ \quad \wedge tr' - tr \in (tr_A^X - tr) \parallel_S (tr_B^X - tr) \\ \quad \wedge \left(\begin{array}{c} ref' \subseteq (ref_A^X \cup ref_B^X) \cap S \cup (ref_A^X \cap ref_B^X) \setminus S \\ \lhd wait' \rhd \\ true \end{array} \right) \end{array} \right)$$

Making use of theorem B.3.2, on page 130, we get:

$$= \exists state, state' \bullet \left(\begin{array}{l} \exists Obs_A^X, Obs_B^X \bullet \\ \quad A_X[Obs_A^X/Obs'] \wedge B_X[Obs_B^X/Obs'] \\ \quad \wedge ok' = ok_A^X \wedge ok_B^X \\ \quad \wedge wait' = wait_A^X \vee wait_B^X \\ \quad \wedge tr' - tr \in (tr_A^X - tr) \parallel_S (tr_B^X - tr) \end{array} \right) \mathbin{;}_X Skip_X$$

Now, using theorem 6.2.2, on page 50, the following equation is acquired.

$$\sqsubseteq \left(\begin{array}{l} \exists Obs_A^C, Obs_B^C \bullet \\ \quad A_C[Obs_A^C/Obs'] \wedge B_C[Obs_B^C/Obs'] \\ \quad \wedge ok' = ok_A^C \wedge ok_B^C \\ \quad \wedge wait' = wait_A^C \vee wait_B^C \\ \quad \wedge ref' \subseteq (ref_A^C \cup ref_B^C) \cap S \cup (ref_A^C \cap ref_B^C) \setminus S \\ \quad \wedge tr' - tr \in (tr_A^C - tr) \parallel_S (tr_B^C - tr) \end{array} \right) \mathbin{;}_C Skip_C$$

This form of the equation makes the proof complete, as the left hand side also has the same form.

Theorem 6.2.6

$$a \rightarrow A_C \sqsubseteq (\exists state, state' \bullet a \rightarrow A_X)$$

Now, we provide the proof for prefixing action.

Proof: Here, we adopt the strategy of reducing the left and right hand side of the equation to the same form. We start the proof by taking the left hand side of the theorem.

$$L.H.S = a \rightarrow_C A_C$$

According to the definition of prefixing action in the *CSP* world, we get:

$$= a \rightarrow_C Skip_C \mathbin{;}_C A_C$$

Now, taking the right hand side of the theorem and solving it to get the expanded form of left hand side.

$$R.H.S = (\exists state, state' \bullet a \rightarrow_X A_X)$$

According to the definition of prefixing action in the *CSP* world, we get:

$$= (\exists state, state' \bullet a \rightarrow_X Skip_X \mathbin{;}_X A_X)$$

Distributing the existensial quantifier $\exists$, the above equation will get the following form:

$$= (\exists state, state' \bullet a \rightarrow_X Skip_X) \mathbin{;}_X (\exists state, state' \bullet A_X)$$

Using the theorem B.1.1, on page 119, will reduce the equation to the following form:

$$= (a \rightarrow_C Skip_C) \mathbin{;}_C (\exists state, state' \bullet A_X)$$

Using theorem 6.2.2, on page 50, makes the proof complete, as the left hand side has also expanded to the same form

$$\sqsubseteq a \rightarrow_C Skip_C \mathbin{;}_C A_C$$

Theorem 6.2.7

$$A_C \setminus_C H \sqsubseteq (\exists state, state' \bullet A_X \setminus_X H)$$

Now, we provide the proof for the hiding construct.

Proof: Here, we adopt the strategy of reducing the left and right hand side of the equation to the same form. We start the proof by taking the right hand side of the theorem.

$$R.H.S = (\exists state, state' \bullet A_X \setminus_X H)$$

According to the definition of hiding in the *Circus* world, we get:

$$= (\exists state, state' \bullet \mathbf{S}(\exists tr_H \bullet A_X[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H) \mathbin{;}_X Skip_X)$$

Now, applying the definition of healthiness condition **CXS4**, we get the following equation:

$$= (\exists state, state' \bullet (\mathbf{CXS4}(\mathbf{S}(\exists tr_H \bullet A_X[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))))$$

Now, taking left hand side of the theorem and solving it to get the expanded form of right hand side.

$$L.H.S = A_C \setminus_C H$$

Applying the definition of hiding construct of *CSP*, we get:

$$= \mathbf{R}(\exists tr_H \bullet A_C[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H) \mathbin{;}_C Skip_C$$

Now, applying the definition of healthiness condition **CSP4**, we get the following equation:

$$= \mathbf{CSP4}(\mathbf{R}(\exists tr_H \bullet A_C[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))$$

According to the defintion of A_C, we get following form:

$$\sqsubseteq \mathbf{CSP4}(\mathbf{R}(\exists tr_H \bullet (\exists state, state' \bullet A_X)[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))$$

As the predicate here does not include $state, state'$, so we can pull $state, state'$ out.

$$= \mathbf{CSP4}(\mathbf{R}(\exists state, state' \bullet \exists tr_H \bullet A_X[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))$$

Using theorem B.2.6, on page 128, will give the following equation:

$$= \mathbf{CSP4}(\exists state, state' \bullet \mathbf{S}(\exists state, state' \bullet \exists tr_H \bullet A_X[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))$$

Now we use theorem B.2.7, on page 129.

$$= \exists state, state' \bullet \mathbf{CXS4}(\exists state, state' \bullet \mathbf{S}(\exists state, state' \bullet \exists tr_H \bullet A_X[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))$$

As $state, state'$ is not mentioned in the inner predicate, we can pull $state, state'$ out.

$$= \exists state, state' \bullet \mathbf{CXS4}(\mathbf{S}(\exists tr_H \bullet A_X[tr_H, H \cup ref'/tr, ref'] \wedge (tr' - tr) = (tr_H - tr) \upharpoonright H))$$

This makes the proof complete, as left and right hand side of the theorem have acquired the same form.

6.3 Summary

In this chapter, we discuss the results required in the mathematical foundation of the translation strategy between the target languages. First of all, the semantics of *Circus* and CSP_M in Unifying Theories of Programming (UTP) framework are specified for the selected constructs of the languages. The chosen subset of the original *Circus* and *CSP* languages are named as Simple*Circus* and Simple*CSP*. The semantical difference between the two is captured by the need for extra observation variables i.e. $state, state'$, in the case of Simple*Circus*. A formal link was proposed to connect the two theories. We propose Conjecture 6.2.1, on page 47 for the linking pairs between the two languages. If P_X is a Simple*Circus* program and P_C is a Simple*CSP* program, then the transformation predicate ($\mathscr{T}$) translates between the two.

For all Simple*Circus* programs P_X, we define a link to the *CSP* world that hides the state. By mathematical proofs included in this chapter, it is proved that this linking predicate preserves the semantics of most of the language operators, with the notable exception of sequential composition. Here, the relationship between $\fatsemi_X$ and $\fatsemi_C$ is proved to be a refinement, not an equality. This is in fact novel work and the major contribution of the thesis. Now, in the following chapter, we describe the translation theory by giving the mathematical representation of the steps involved in the translation process.

Chapter 7

Translation Theory

This chapter describes the formalised version of the translation process. The translation process is divided into a number of steps. The action bodies of Simple*Circus* processes are gone through these steps in order to translate to their equivalent Simple*CSP* output.

7.1 The Template for Each Function in the Translation Process

The following is a common overlapping syntax for Simple*Circus* and Simple*CSP*

$$
\begin{array}{rcll}
P \in Process & ::= & A\,\mathbf{where}\,D_1;\ \ldots;\ D_n & \\
D \in ActDef & ::= & N \mathrel{\widehat=} A \mid N(x_1,\ldots,x_n) \mathrel{\widehat=} A & \\
A \in Action & ::= & Stop \mid Skip \mid a \to A \mid e \,\&\, A \mid A \fatsemi A \mid A \setminus H & \text{— sequential subset} \\
 & & \mid A \sqcap A \mid A \,\Box\, A & \\
 & & \mid A \parallel_S A & \text{— } \mathit{CSP} \text{ only} \\
 & & \mid x := e \mid A[_V|_H|_V]A & \text{— } \mathit{Circus} \text{ only} \\
 & & \mid N \mid N(e_1,\ldots,e_n) & \\
 & & \mid \mathscr{C}_i(e_1,\ldots,e_n) & \text{— translation temporary only} \\
 & & \mid \fatsemi_i & \text{— translation temporary only} \\
N \in Name & ::= & \text{names} & \\
a, b.e \in Event & & \text{events} & \\
e \in Expr & & \text{expressions} & \\
x \in Var & & \text{variables} & \\
V \in VarSet & & \text{variable-sets} & \\
\mathscr{C} \in ContName & & \text{continuation names} & \\
i \in Indices & & \text{an index set} &
\end{array}
$$

A similar version of this mixed syntax was introduced in figure 1.2 of Chapter 1. The difference between figure 1.2 and this syntax is the inclusion of *translation temporary placeholders*. These temporary placeholders are needed during the appliction of translation steps, discussed in the preceding section.

The formalised translation process is divided into steps. Each step is implemented as a function. If f is a function then it is applied to each language construct. The function f takes an input of type *Action*, and possibly some extra parameters, and returns a result/output, also of type *Action*, also possibly with some extra outcome data..

$$
\begin{array}{rcl}
f & : & Action \to Action \\
f(Input) & \widehat= & Output
\end{array}
$$

For example, in our translation process, the first step is called "Normalise Sequential Composition". This step is implemented as a function, abbreviated as *NSC*. The application of this function to an action will result in some output action, as shown.

$$\begin{array}{rcl} NSC & : & Action \rightarrow Action \\ NSC(Stop) & \widehat{=} & SomeOutputFromNSC \\ NSC(Skip) & \widehat{=} & SomeOutputFromNSC \\ NSC(a \rightarrow A) & \widehat{=} & SomeOutputFromNSC \\ & & \text{and so on.} \end{array}$$

In short, each translation step is a function from *Action* to *Action*, typically defined by pattern matching against the range of syntax forms.

7.2 The Translation Process

7.2.1 Overview

Given a Simple*Circus* process description

$$A \, \mathbf{where} \, D_1; \, \ldots; \, D_n$$

we translate as follows:

Normalise Sequential Composition

Transform all nested uses of $\fatsemi$ into right associative forms, and apply the Skip unit laws to eliminate as many uses of *Skip* as possible.

Rename Hidden Events

Rename hidden events to avoid clashes, using the following law:

$$P \setminus \vec{h} = P[\vec{n}/\vec{h}] \setminus \vec{n}$$

where $\vec{h}$ is a canonically ordered set (H) of events and $\vec{n}$ are globally fresh.

Name "Next" Actions

For every sequential composition $P \fatsemi Q$, if Q is not atomic (*Stop*, *Skip*, *Chaos* or a call $N(\ldots)$) then we introduce a fresh name N_Q and definition $N_Q \widehat{=} Q$, and replace the composition by $P \fatsemi N_Q$.

Get Variable Parameters

Gather all assignment variables and extend parameter lists of all definitions to include these. We assume that pre-existing call parameter list names do not overlap with assignment variables.

Ensure Assignment Continutation

Every assignment $x := e$ that is not immediately followed by $\fatsemi$ we replace with $x := e \fatsemi Skip$.

Add Continuation Calls

Replace every *Skip* with a call to a fresh distinguished name $\mathscr{C}_x$, and every arm of every choice gets a fresh $\mathscr{C}_y$ sequentially composed on the end.

Propagate Assignments

Propagate the assignment effects up as far as the next call (including any $\mathscr{C}_x$), removing the assignments in passing.

Instantiate Continuations

For every sequential composition $P \fatsemi_x N(\ldots)$, create a new instance of P in which every $\mathscr{C}_x(e_1,\ldots,e_2)$ is replaced by $N(e_1,\ldots,e_2)$.

We shall adopt the following shorthand in the sequel to reduce mathematical boilerplate:

- Unary *Circus* constructors: $a \rightarrow A$, $e \,\&\, A$ and $A \setminus H$ will be covered by the notation $F\,A$.

- Binary *Circus* constructors: $A \sqcap B$, $A \,\Box\, B$, $A[_U|_S|_V]B$, $A \fatsemi B$ will be covered by the notation $A \oplus B$.

7.2.2 Normalise Sequential Composition

Transform all nested uses of $\fatsemi$ into right associative forms, and apply the Skip unit laws to eliminate as many uses of *Skip* as possible.

$$\begin{array}{rcl}
NSC & : & ActDef^* \rightarrow ActDef^* \\
NSC(Skip \fatsemi c) & \widehat{=} & NSC(c) \\
NSC(c \fatsemi Skip) & \widehat{=} & NSC(c) \\
NSC(c_1 \fatsemi c_2) & \widehat{=} & NSC(c_1) \fatsemi_R NSC(c_2) \\
NSC(F\,A) & \widehat{=} & F\,NSC(A) \\
NSC(A \oplus B) & \widehat{=} & NSC(A) \oplus NSC(B) \\
NSC(A) & \widehat{=} & A \quad \text{—when none of the above apply\ldots}
\end{array}$$

$$\begin{array}{rcl}
(c_1 \fatsemi (c_2 \fatsemi c_3)) \fatsemi_R c_4 & \widehat{=} & c_1 \fatsemi ((c_2 \fatsemi c_3) \fatsemi_R c_4) \\
(c_1 \fatsemi c_2) \fatsemi_R c_3 & \widehat{=} & c_1 \fatsemi (c_2 \fatsemi c_3) \\
c_1 \fatsemi_R c_2 & \widehat{=} & c_1 \fatsemi c_2
\end{array}$$

7.2.3 Rename Hidden Events

Rename hidden events to avoid clashes, using the following law:

$$P \setminus \vec{h} = P[\vec{n}/\vec{h}] \setminus \vec{n}$$

where $\vec{h}$ is a canonically ordered set (H) of events and $\vec{n}$ are globally fresh.

For now, we do not elaborate further, but rather assume that all hidden events are disjoint from non-hidden ones, at every level of the hierarchy This rather avoids the benefits of hiding in name-space management, but is just a piece of boiler-plate “alpha-substitution” that is of little interest at present. We also assume that this includes α-renaming all existing parameters so they do not clash with assignment variables.

7.2.4 Name “Next” Actions

For every sequential composition $P \fatsemi Q$, if Q is not atomic (*Stop*, *Skip*, *Chaos* or a call $N(\ldots)$) then we introduce a fresh name N_Q and definition $N_Q \widehat{=} Q$, and replace the composition by $P \fatsemi N_Q$.

$$
\begin{array}{rcl}
NNA & : & Action \rightarrow Action \times ActDef^* \\
\textbf{let } (A', \delta') & = & NNA(A) \\
(B', \delta'') & = & NNA(B) \\
\textbf{below} & & \\
NNA(A \mathbin{;} N) & \widehat{=} & (A' \mathbin{;} N, \delta') \\
NNA(A \mathbin{;} B) & \widehat{=} & (A' \mathbin{;} N_B, \delta' \frown \delta'' \frown \langle N_B \widehat{=} B' \rangle), N_B \text{ fresh} \\
 & & \text{—rest is boilerplate...} \\
NNA(F A) & \widehat{=} & (F A', \delta') \\
NNA(A \oplus B) & \widehat{=} & (A' \oplus B', \delta' \frown \delta'') \\
NNA(A) & \widehat{=} & (A, \langle\rangle) \quad \text{—when none of the above apply...}
\end{array}
$$

7.2.5 Get Variable Parameters

Gather all assignment variables and extend parameter lists of all definitions to include these. We assume that pre-existing call parameter list names do not overlap with assignment variables.

This stage is sub-divided into three phases.

$$
\begin{array}{rcl}
GVP & : & ActDef^* \rightarrow ActDef^* \\
GVP(P) & \widehat{=} & GVP3 \circ GVP2 \circ GVP1
\end{array}
$$

The first phase simply accumulates variables used in each action in isolation, and notes any calls to other actions.

$$
\begin{array}{rcl}
GVP1 & : & ActDef \rightarrow ActDef \times \mathcal{P}\,Var \times \mathcal{P}\,Name \\
GVP1(A) & \widehat{=} & (A, COLLV(body.A), COLLC(body.A))
\end{array}
$$

$$
\begin{array}{rcl}
COLLV & : & Circus \rightarrow \mathcal{P}\,Var \\
COLLV(x := e) & \widehat{=} & \{x\} \\
COLLV(F A) & \widehat{=} & COLLV(A) \\
COLLV(A \oplus B) & \widehat{=} & COLLV(A) \cup COLLV(B) \\
COLLV(A) & \widehat{=} & \emptyset \quad \text{—when none of the above apply...}
\end{array}
$$

$$
\begin{array}{rcl}
COLLC & : & Circus \rightarrow \mathcal{P}\,Name \\
COLLC(N(\ldots)) & \widehat{=} & \{N\} \\
COLLV(F A) & \widehat{=} & COLLV(A) \\
COLLV(A \oplus B) & \widehat{=} & COLLV(A) \cup COLLV(B) \\
COLLV(A) & \widehat{=} & \emptyset \quad \text{—when none of the above apply...}
\end{array}
$$

The second phase follows the call patterns, effectively computing a form of transitive closure: an action uses variables in those actions it calls as well as its own.

$$
\begin{array}{rcl}
GVP2 & : & (ActDef \times \mathcal{P}\,Var \times \mathcal{P}\,Name)^* \rightarrow (ActDef \times \mathcal{P}\,Var \times \mathcal{P}\,Name)^* \\
GVP2 & & \text{— if A calls B extend both with each others vars, until no change}
\end{array}
$$

The third and final phase simply extends each definition parameter list with the variables discovered in the previous two phases.

$$
\begin{aligned}
GVP3 &: (ActDef \times \mathscr{P}\,Var \times \mathscr{P}\,Name)^* \rightarrow ActDef \\
GVP3(N \mathrel{\widehat{=}} A, \nu, _) &\mathrel{\widehat{=}} (N(\nu) \mathrel{\widehat{=}} ADDV_\nu(A))
\end{aligned}
$$

$$
\begin{aligned}
ADDV &: Var^* \rightarrow Action \rightarrow Action \\
\textbf{let}\ A' &= ADDV_\nu(A) \\
B' &= ADDV_\nu(B) \\
\textbf{below} & \\
ADDV_\nu(N(\ldots)) &\mathrel{\widehat{=}} N(\ldots, \nu) \\
ADDV_\nu(F\,A) &\mathrel{\widehat{=}} F\,A' \\
ADDV_\nu(A \oplus B) &\mathrel{\widehat{=}} A' \oplus B' \\
ADDV_\nu(A) &\mathrel{\widehat{=}} A \quad \text{—when none of the above apply\ldots}
\end{aligned}
$$

7.2.6 Ensure Assignment Continutation

Every assignment $x := e$ that is not immediately followed by $\fatsemi$ we replace with $x := e \fatsemi Skip$.

$$
\begin{aligned}
EAC &: Var^* \rightarrow Action \rightarrow Action \times ActDef \\
EAC_\nu(x := e \fatsemi B) &\mathrel{\widehat{=}} x := e \fatsemi EAC_\nu(B) \\
EAC_\nu(A \fatsemi B) &\mathrel{\widehat{=}} EAC_\nu(A) \fatsemi EAC_\nu(B) \\
EAC_\nu(x := e) &\mathrel{\widehat{=}} x := e \fatsemi Skip \\
EAC_\nu(F\,A) &\mathrel{\widehat{=}} F\,EAC_\nu(A) \\
EAC_\nu(A \oplus B) &\mathrel{\widehat{=}} EAC_\nu(A) \oplus EAC_\nu(B) \\
EAC_\nu(A) &\mathrel{\widehat{=}} A \quad \text{—when none of the above apply\ldots}
\end{aligned}
$$

7.2.7 Add Continuation Calls

Replace every *Skip* with a call to a fresh distinguished name $\mathscr{C}_x$, and every arm of every choice gets a fresh $\mathscr{C}_y$ sequentially composed on the end.

$$
\begin{aligned}
ACC &: Var^* \rightarrow Action \rightarrow Action \times ActDef \\
\textbf{let}\ (A', \alpha) &= ACC_\nu(A) \\
(B', \beta) &= ACC_\nu(B) \\
\textbf{below} & \\
ACC_\nu(Skip) &\mathrel{\widehat{=}} (\mathscr{C}_x(\nu), \langle \mathscr{C}_x(\nu) \mathrel{\widehat{=}} Skip \rangle), \quad x \text{ fresh.} \\
ACC_\nu(A \sqcap B) &\mathrel{\widehat{=}} (A' \fatsemi \mathscr{C}_x(\nu) \sqcap B' \fatsemi \mathscr{C}_y(\nu), \alpha \frown \beta \frown \langle \mathscr{C}_x(\nu) \mathrel{\widehat{=}} Skip \rangle \frown \langle \mathscr{C}_y(\nu) \mathrel{\widehat{=}} Skip \rangle), x, y \text{ fresh} \\
ACC_\nu(A \,\square\, B) &\mathrel{\widehat{=}} (A' \fatsemi \mathscr{C}_x(\nu) \,\square\, B' \fatsemi \mathscr{C}_y(\nu), \alpha \frown \beta \frown \langle \mathscr{C}_x(\nu) \mathrel{\widehat{=}} Skip \rangle \frown \langle \mathscr{C}_y(\nu) \mathrel{\widehat{=}} Skip \rangle), x, y \text{ fresh} \\
ACC_\nu(F\,A) &\mathrel{\widehat{=}} (F\,A', \alpha) \\
ACC_\nu(A \oplus B) &\mathrel{\widehat{=}} (A' \oplus B', \alpha \frown \beta) \\
ACC_\nu(A) &\mathrel{\widehat{=}} (A, \langle\rangle) \quad \text{—when none of the above apply\ldots}
\end{aligned}
$$

7.2.8 Propagate Assignments and Instantiate Continuations

Propagate Assignments: Propagate the assignment effects up as far as the next call (including any $\mathscr{C}_x$), removing the assignments in passing.
Instantiate Continuations: For every sequential composition $P \, ⨾_x \, N(\ldots)$, create a new instance of P in which every $\mathscr{C}_x(e_1,\ldots,e_2)$ is replaced by $N(e_1,\ldots,e_2)$.

A first attempt, noting that we need to manage assignments in context, e.g. being aware of subtle syntactic differences like $(a \rightarrow x := 1);\ N(x,\ldots)$ and $a \rightarrow (x := 1;\ N(x,\ldots))$, that should both translate to the same outcome: $a \rightarrow N(1,\ldots)$.

$$
\begin{array}{rcl}
PA & : & (Var \nrightarrow Expr) \rightarrow Action \rightarrow ((Var \nrightarrow Expr) \times Action) \\
\textbf{let } (\sigma',A') & = & PA_\sigma(A) \\
(\sigma'',B') & = & PA_\sigma(B) \\
\varsigma & = & \sigma \dagger \{x \mapsto e\} \\
\textbf{below} & & \\
PA_\sigma(x := e) & \widehat{=} & (\varsigma, Skip) \\
PA_\sigma(\mathscr{C}(e_1,\ldots,e_n)) & \widehat{=} & (\sigma, \mathscr{C}(e_1\sigma,\ldots e_n\sigma)) \\
PA_\sigma(e \,\&\, A) & \widehat{=} & (\sigma', e\sigma \,\&\, A') \\
PA_\sigma(F\,A) & \widehat{=} & (\sigma', F\,A')(\sigma', A \setminus H) \\
PA_\sigma(A \,⨾\, N(e_1,\ldots,e_n)) & \widehat{=} & (\sigma', A' \,⨾\, N(e_1\sigma',\ldots,e_n\sigma')) \\
PA_\sigma(A[U|S|V]B) & \widehat{=} & (\sigma'[U|S|V]\sigma'', PA_\sigma(A)[U|S|V]PA_\sigma(B)) \\
PA_\sigma(N(e_1,\ldots,e_n)) & \widehat{=} & (\sigma, N(e_1\sigma,\ldots,e_n\sigma)) \\
 & & \\
PA_\sigma(A \sqcap B) & \widehat{=} & (\sigma' \uplus \sigma'', (PA_\sigma(A) \sqcap PA_\sigma(B)) \,⨾\, M(i,\nu_1,\nu_2)) \\
PA_\sigma(A \,\Box\, B) & \widehat{=} & (\sigma' \uplus \sigma'', (PA_\sigma(A) \,\Box\, PA_\sigma(B)) \,⨾\, M(i,\nu_1,\nu_2)) \\
 & & \\
PA_\sigma(A) & \widehat{=} & (\sigma, A) \quad \text{—when none of the above apply\ldots}
\end{array}
$$

Here $e\sigma$ denotes e with substitution done as per σ, and $\nu\sigma$ means each x in ν is so substituted. In addition, $\dagger$ is function override.

In the case of internal and external choice, $\uplus$ is disjoint union that distinguishes the left and right sides by tagging the variables. A suitable tagging mechanism ensures that if expression e is meant to be substituted in left side variables and expression f is meant to be substituted in the right side, the correct substitution is performed. Mathematically, if we have,

$$v \nrightarrow e \uplus v \nrightarrow f$$

It becomes:

$$v_L \nrightarrow e, v_R \nrightarrow f$$

Here, the call $M(i, v_L, v_R)$ ensures the final continuation call uses the appropriate variable set.

Here we used a single $\mathscr{C}$ process, but it now looks like we need to tag them to link to names after specific $⨾$s.

So

$$(\ldots \mathscr{C}_x \ldots) \,⨾_x\, N$$

will become

$$(\ldots N \ldots)$$

in step "Instantiate Continuations".

Consider the following:

$$(\ldots(\ldots Skip \fatsemi N \ldots)\ldots Skip) \fatsemi M \ldots$$

If we do not distinguish *Skip*s and $\fatsemi$s, then we get

$$(\ldots(\ldots \mathscr{C} \fatsemi N \ldots)\ldots \mathscr{C}) \fatsemi M \ldots$$

and Instantiate Continuations risks doing incorrect replacements to obtain, for instance:

$$(\ldots(\ldots M \ldots)\ldots M)\ldots$$

With proper labelling, shown below:

$$(\ldots(\ldots \mathscr{C}_1 \fatsemi_1 N_1 \ldots)\ldots \mathscr{C}_2) \fatsemi_2 M_2 \ldots$$

"Instantiate Continuations" can get it right:

$$(\ldots(\ldots N \ldots)\ldots M)\ldots$$

In order to elaborate more on the translation for internal and external choice, we take a running example from SimpleCircus2CSPM. The *Circus* specification of this example is as follows:

```
A ^= a -> (x := 1) ; Skip

B ^= (y := x+1) ; b -> Skip

C ^= c -> Skip

MAIN ^= (z := 3) ; (A() ; B() |~| A() ; C())
```

The translated version of this particular example is:

```
A(x,y,z) ^= a -> Skip

A_2(x,y,z) ^= a -> MAIN_3(1,y,z)

B(x,y,z) ^= B_2(x,x+1,z)

B_2(x,y,z) ^= b -> Skip

C(x,y,z) ^= c -> Skip

MAIN(x,y,z) ^= MAIN_2(x,y,3)

MAIN_2(x,y,z) ^= A_2(x,y,z)

MAIN_3(x,y,z) ^= B(x,y,z) |~| C(x,y,z)
```

Here, the main function first sets the variable z to a value. This assignment command is followed by an internal choice between left and right sequentially composed chains of actions. Effectively, there is an internal choice between actions *B* and *C*. In the translated version, the proper tagging for the call of actions is performed. Furthermore, the variables are substituted with their expressions correctly.

7.3 Summary

Now, we wrap up this chapter. The translation process from *Circus* to CSP_M is divided into steps. In fact, each translation step is a function, typically defined by pattern matching against the range of syntax forms. These steps are formalised and mathematically represented here. Once formally specified, the mathematical representation made the implementation phase much easier and straightforward, particularly in the case of developing the SimpleCircus2CSPM prototype in Haskell. The following chapter gives the implementation details of both the circus2cspm and SimpleCircus2CSPM tools.

Chapter 8

Implementation

This chapter provides the information on the implementation of the translation between *Circus* and CSP_M. First, it describes our initial attempt of translating between *Circus* and CSP_M using Java *i.e.*, the conversion of JCircus sources to circus2cspm. Then, we describe the implementation of the translator in Haskell.

8.1 Implementation Initial Attempt – JCircus to circus2cspm

In the figure 8.1, the steps for the translation process of our translator circus2cspm are described informally. These steps are similar to those mentioned in section 5.2.1 to 5.2.4. The flow diagram is a representation of the initial setup of steps for translation process, implemented in circus2cspm tool.

The first step is about specification file loading. Here, we gather the information about the channel declarations. Then, we gather the state variables defined in Z schemas. From these, we develop a list of parameters, to be attached to each process name. The second step involves the analysis of the main action. Here, we establish a record of the calling actions and the operators in the initialiser. Now, each *CSP*-like action in the *Circus* specification is converted into its equivalent machine readable version. If an assignment command appears, the expression is replaced at the appropriate place in the parameter list, attached to corresponding process name. This gives us an intermediate translated version of the specification. The final translation step involves the replacement of the correct calling actions in the intermediate translated version. The replacement is performed on the basis of analysis of main action done in Step 2 of the translation.

8.1.1 The Explanation of the Translation Flow Diagram with a Simple Example

Our formal description of the translation strategy below states that in order to translate a program in *Circus* to *CSP*, we get the channels declared in the program. After that, we obtain the schemas defined in the program and extract from them a list of state variables. Here 'g' in `gstatevars` stands for global. The next step is to load all the process paragraphs in the program.

The process paragraphs are composed of number of state variables and *Circus* actions. After developing the list of parameters from the state variables, the description of each individual *Circus* action is gathered. The purpose of initialiser part of the *Circus* actions in a process paragraph is to establish the order of the *Circus* actions performed in the main action.

The translation is then a matter of converting the gathered information of the Circus process into its machine readable version while performing transformation steps on the gathered channels, state variables and process paragraphs information.

```
translate prog
```

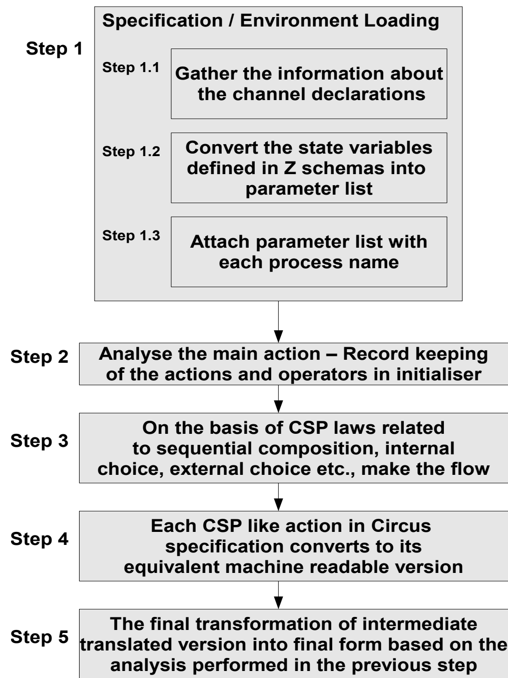


Figure 8.1: Generic Translation Process Flow

```
   = let chans = getchanns prog in
     let gstatevars = get schemas prog in
     let circprocess = get process prog in
     let CSPprocess = get translateProc chans gstatevars prog
     in output CSPprocess

translateproc chans gstatevars proc
   = let statevars = (getschemas proc) ++ gstatevars in
     let params = metaparas statevars in
     let circactions = getactions proc in
     let CSPactions = map (tanslateAct chans params) circactions
     let InitCSP = translateAct chans params (actions | proc)
       in CSPassemble CSPactions, InitCSP
```

In order to explain each step of the translation process, we use a *Circus* program given in figure 8.2, to elaborate the input and output of each step of the translation. This example contains channel declaration and a process definition with name $Ex1$. The Z schema has two state variables x and y. The actions are constituted of assignment commands, sequential composition, prefix action and one of the basic actions, that is, *Skip*. The main action defined is kept simple with all sequential compositions between each call of a particular action.

8.1.2 Step 1 – Specification/Environment Loading and Information Gathering

Section 4.2.3 of [dF05] describes the environment setting of the tool formally. For example, the formal description for the channels environment is described as $\mathsf{VisChanEnv} : \mathsf{seq}\ \mathbb{N}$, recording the names of visible channels of a process. $\mathsf{HidChanEnv} : \mathsf{seq}\ \mathbb{N}$ records the names of hidden channel of a process. SyncCom-

channel a, b, c
process $Ex1 \widehat{=}$ **begin**
state

$Ex1State$
$x, y : \mathbb{N}$

$ACTION_A \widehat{=} x := 1;\ a \rightarrow Skip$
$ACTION_B \widehat{=} y := x + 1;\ b \rightarrow Skip$
$ACTION_C \widehat{=} c \rightarrow Skip$
$\bullet\ ACTION_A;\ ACTION_B;\ ACTION_C$
end

Figure 8.2: *Circus* Example to Explain Translation Process

$\mathsf{mEnv} : \mathsf{N} \nrightarrow \mathsf{SC}$ maps each channel name used in a process to a value of type $\mathsf{SC} : \mathsf{S}$ (if synchronisation channel) or $\mathsf{SC} : \mathsf{C}$ (if communication channel). A communication channel is the channel which inputs (indicated by '?') or outputs (indicated by '!') a value on the channel. Meanwhile, the synchronisation channel is an untyped channel or it contains fields that are not defined as input or output.

Now, in the following, we describe the steps of the translation process:

8.1.3 Step 1.1 – Getting Channel Information

In this step, the task is to gather the information of channel declarations, CDecls, in the loaded specification, Program.

$$\text{Step}\,1.1 : \mathsf{Program} \rightarrow \mathsf{CDecls}^*$$

We process a list of process paragraphs, Paras, having empty set of CDecls. Each Para we process may add some more channel declarations.

$$\text{Step1.1Para} : \mathsf{CDecls}^* \rightarrow \mathsf{Para} \rightarrow \mathsf{CDecls}^*$$
$$\text{Step1.1Para}\ \mathsf{CDecls}\ \mathsf{CDecl}\ = \mathsf{CDecl} \mathbin{++} [\mathsf{CDecls}]$$
$$\text{Step1.1Para}\ \mathsf{CDecls}\ \mathsf{Para}\ = \mathsf{CDecls}$$

If we take the *Circus* example, $Ex1$ mentioned earlier, after completing this step, the translated version of CSP_M will be:

```
channel a
channel b
channel c
```

8.1.4 Step 1.2 – Process the Z schemas to know the state variables

The next step is to gather the information of the state variables defined in a process. After gathering this data, the state variables defined in Z schemas are converted into a parameter list. As in a *Circus* specification, the Z is mixed freely within the specification, so the new state variables can be added anywhere in the *Circus*

specification. Consequently, the parameter list is required to add in the upcoming variables into the existing list during parsing. We process a list of process paragraphs, Paras start with an empty set of parameters, Params. Each Para we process, may add some new parameter into the parameter list, appended in Params.

$$\text{Step1.2Para} : \mathsf{ParamList} \rightarrow \mathsf{Para} \rightarrow \mathsf{ParamList}$$
$$\text{Step1.2Para}\ \mathsf{Params}\ \mathsf{zschema} = \mathit{paramsOf}\ \mathsf{zschema} ++ [\mathsf{Params}]$$
$$\text{Step1.2Para}\ \mathsf{Params}\ \mathsf{Para} = \mathsf{Params}$$

The function *paramsOf* is used here is to extract the state variables from zschema. After completing these two sub-steps for a process N now we know the declared channels and the state variables.

In order to elaborate more, if we process the schema defined in our *Circus* example $Ex1$, the following schema will turn into a parameter list of (x, y).

$Ex1State$

$x, y : \mathbb{N}$

In Java code, this step is achieved by calling `visitingSignature` function in `TranslatorVisitor` class. The function `visitingSignature` is called from the `visitBasicProcess` function. A local function named `variableListFunction` is provided to retrieve list of variables at any time of execution. A globally scoped list of strings named `state_variables` is provided to store the schema information into a parameter list.

8.1.5 Step 1.3 – Making each action to its parameterised version

This step of the translation involves attaching the parameter list to each process name. This step is necessary because in CSP_M, each of the processes is called by passing the values of the parameters. From an implementation prospective, this functionality is achieved by the globally defined list of strings named `state_variables`. This list has already gathered the information of parameters available in the process in step 1.2. So, now attaching the list of parameters to a function name is just a matter of outputting this list into the required form at the required places.

8.1.6 Step 2, 3 – Analysis of the Main Action

The *Skip* in an individual action is replaced by a continuation marker i.e. CONT_, followed by the particular action name.

We use the process initialiser part of the specification, i.e. *Init* to see the common pattern, e.g. in our particular case, analysing A ; B ; C turns into A ⇒ B ⇒ C, i.e. A will be followed by B and B will be followed by C. More closely, if A = ... CONT_A(x,y) and A feeds into B then translation would be: A(x,y) ... → B(x,y)

In our simple example $Ex1$, the actions are followed by one another i.e. the main action contains all sequential actions e.g *Action_A*; *Action_B*; *Action_C*. Here, we keep record of the action names in the main action and the operators involved. We used two lists `rkActionsInMain` and `rkOperatorsInMain` for this purpose. After visiting the main action: • *Action_A*; *Action_B*; *Action_C*, the contents of `rkActionsInMain` and `rkOperatorsInMain` contents' are shown in the figure 8.3 below.

When the assignment command in the actions i.e. *Action_A*, *Action_B* and *Action_C* are visited, the record keeping of the corresponding expressions for the variable list is recorded in the global scope list named `rkExprsForSkip`. For uniquely identifying the expressions for each action, the respective action name is

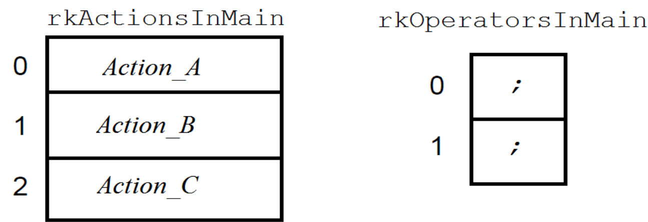


Figure 8.3: Record Keeping for Actions Called in Main Action and Linking Operators

also stored on the first index, followed by the expressions. To elaborate this, the content of `rkExprsForSkip` is shown in the figure 8.4 when the assignment command $x := 1$ and $y := x + 1$ have been visited respectively.

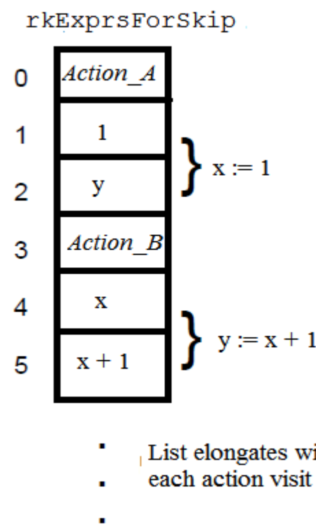


Figure 8.4: Record Keeping for Expressions when visiting Assignment Command

Furthermore, the length of the list of expressions for a particular action is dependent on the number of parameters defined for a particular process, that is, in the case of our simple example, as x and y are the state variables defined in the specification. So, the length of list of expressions for a particular action would be equal to 2.

When the function `visitSkipAction` is called, initially the *Skip* will be replaced by the CONT_ followed by the action name. This information is stored in the global scope list called `rkContinueActionsForSkip`. This replacement of *Skip* with the Continue functions is depicted below in the figure 8.5.

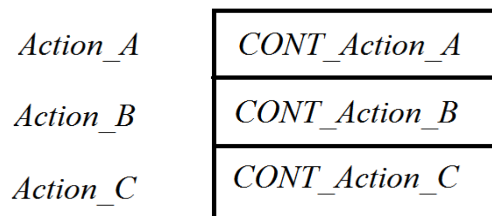


Figure 8.5: Record Keeping for Continue Functions when visiting Skip Action

8.1.7 Step 4 – Conversion of *CSP*-like actions to CSP_M and Alignment of Information Gathered So Far

In step 4, each of the *CSP*-like actions in the *Circus* specification converts to its equivalent machine readable version.

Now in order to elaborate the conversion in this step, the intermediate format of our *Circus* example $Ex1$ should turn into the following form:

$$\begin{aligned} Action_A(x,y) &= a \rightarrow CONT_Action_A(1,y) \\ Action_B(x,y) &= b \rightarrow CONT_Action_B(x,x+1) \\ Action_C(x,y) &= c \rightarrow CONT_Action_C(x,y) \end{aligned}$$

When the function `visitingPrefixAction` is called during the specification translation, the information gathered so far is aligned and a string is output for this intermediate stage. One functionality is gathering information about the prefixed channels related to a particular action, e.g. in our example the prefixed channel for *ACTION_A* will be 'a'. At this stage of the translation, the aligned information is gathered together as a string named `outputStringPrefixAction` for a particular action. To uniquely identify the string related to a particular action, the action name is indexed first in a list of output strings followed by the respective string as shown in figure 8.6.

ACTION_A	a->	CONT_ACTION_A	1	y
ACTION_B	b->	CONT_ACTION_B	x	x+1
ACTION_C	c->	CONT_ACTION_C	x	y

Pieces of Information in Bits

ACTION_A	a->CONT_ACTION_A(1,y)
ACTION_B	b->CONT_ACTION_B(x,x+1)
ACTION_C	c->CONT_ACTION_C(x,y)

Aligned Information in `outputStringPrefixAction`

Figure 8.6: Record Keeping for Output Strings from Prefix Action

8.1.8 Step 5 - Final Transformation

The final step in the translation is replacing the CONT_ACTION function by the following action appearing in the list of action names found in the main action call. We have already stored this sequence in the global list named `rkActionsInMain`. Here the CONT_ACTION function is an indication that a *Skip* was there. So one can immediately follow the pattern found in the list of recorded action names i.e. `rkActionsInMain` and replace the CONT_ with the following action name. In the case of the last string the *Skip* will simply be replaced by SKIP. The figure 8.7 elaborates the final step of translation.

So, final translated version of the *CSP* code will be:

```
ACTION_A(x,y) = a -> ACTION_B(1,y)
ACTION_B(x,y) = b -> ACTION_C(x,x+1)
ACTION_C(x,y) = c -> SKIP
```

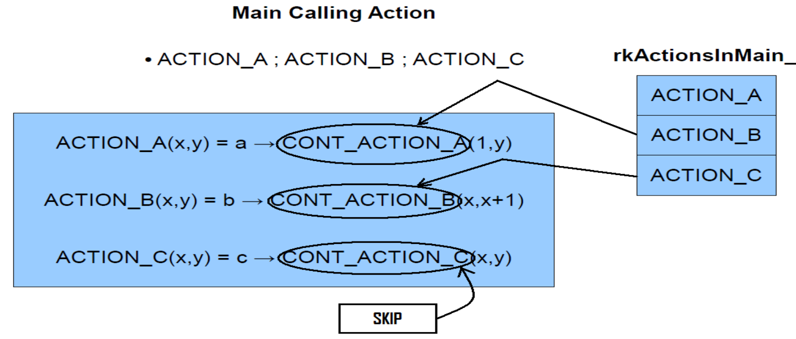


Figure 8.7: Final Replacement Step on the basis of Main Calling Action

Now, if we consider the case of a changed sequence of actions in the main action call, the final translation step must follow the sequence appearing in the main action call. For example, if the sequence in the main action call is now • *Action_C*; *Action_B*; *Action_A*, then the final translation step of replacing continue actions will be as shown in the figure 8.8.
Now in this case, final translated version of the *CSP* code will be:

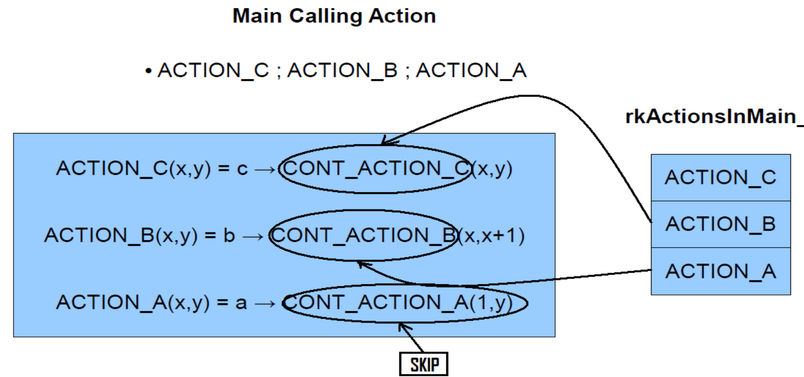


Figure 8.8: Final Replacement Step with Changed Sequence of Actions

```
ACTION_C(x,y) = c -> ACTION_B(x,y)
ACTION_B(x,y) = b -> ACTION_A(x,x+1)
ACTION_A(x,y) = a -> SKIP
```

Here the Simple*Circus* process main contains is containing only sequential composition. The laws of *CSP* have to be followed while dealing with other language constructs like $\Box$, $\sqcap$, $\|$, etc.

A number of examples translated from circus2cspm are discussed in the evaluation chapter in Section 9.3 on page 93.

Although we have achieved the correct output of a number of translation examples for a subset of constructs from *Circus* to CSP_M, however, we found the development in Java to be quite complex, especially in the record keeping process and moving from initial to intermediate and then from intermediate to the final transformation of the translated version. We dealt here with some simple action compositions. But if this structure grows and acquires a complex form, it would be quite hard to implement it. Pattern matching in the compositions of complex combinations of actions in the main action is hard to find and implement in the present state of the source code of circus2cspm. This pattern matching is also a pre-requisite while dealing with the implementation of all *CSP* laws in the main action. This goal was found to be very hard to achieve as an implementation in Java.

As an alternative approach, the use of Haskell, a well-known functional programming language, for modelling these translations and acquiring *CSP* normal form was found to be much easier. This implementation can be achieved in far less lines of code than its equivalent in Java. The use of functional languages is quite easy when dealing with pattern matching. In comparison to few lines of code in Haskell, the `TranslatorVisitor` class size was increased from 3850 to 4987 lines after the implementation of few combinations of action calls in main action, as described above.

Now we give the implementation details of the development of a prototype in Haskell.

8.2 Haskell Implementation – SimpleCircus2CSPM

This section gives a step-by-step explanation of prototyping the translation strategy in Haskell. Portions of Haskell code are included for the prototype details.

8.2.1 Names

First of all, we defined all the names involved in our target languages. The languages are kept simple and so the proposed name for the language is Simple*Circus*. It covers a subset of *Circus* and CSP_M constructs. The data type of the names is `String`. The names for binary operators of `not`, `or`, `and`, `equal` or `implies` are given. Similarly, standard operators for comparison, arithmetic operations are given relative names. The set notations involve the names for usual sets and channel sets. The operations on sets are union ($\cup$), intersect ($\cap$), difference (`\\`) and subset ($\subset$). The sequences involve their definition, catenation of two sequences and determining the length by operator `#`. The names specific to Simple*Circus* are: interleaving ($|||$), alphabetised parallel ($_X||_Y$), interface parallel ($||_X$), internal choice ($\sqcap$), external choice ($\Box$), sequential composition (;), guard (&), prefix operator ($\rightarrow$), hiding ($\backslash$), assignment (`:=`) and recursion (`Mu`). The type for prototyping names in Haskell is:

```
type Name = String
```

Listed below are the *Circus* specific names.

```
ilvName = "|||"
interfaceParOpenName = "[|"
interfaceParCloseName = "|]"
alphabetisedParOpenName = "["
alphabetisedParMidName = "||"
alphabetisedParCloseName = "]"
repAlphabetisedOpenName = "["
repAlphabetisedCloseName = "]"
icName = "|~|"
ecName = "[]"
seqName = ";"
guardName = "&"
prefixName = "->"
hideName = "\\"
atName = "@"
replName = "RR"
ifName = "if"
thenName = "then"
elseName = "else"
```

```
asgName = ":="
muName = "Mu "
```

8.2.2 Expressions

In Simple*Circus*, we modelled expressions as a boolean, an integer, a variable name, an aggregate function which aggregates a list of expressions, application of name to an expression list, combining two expressions in one name using the `Bin` function and depicting events on a channel. The interesting one is modelling of events which are defined to be of type `EvtSpec` (event specfication). The event communication can be a Null; inputting an expression on a channel or outputting an expression on a channel. Here `[Comm]` lists a number of communications on a channel. This is defined in Haskell as:

```
type EvtSpec = (Name, [Comm])

data Comm
 = Null | Dot Expr | Bang Expr | Q Name (Maybe Expr)
 deriving (Eq, Ord, Show)

data Expr
 = B Bool | Z Int | Var Name
 | Agg Name Name [Expr]
 | App Name [Expr]
 | Bin Name Expr Expr
 | Evts EvtSpec
 deriving (Eq,Ord,Show)
```

8.2.3 Expression Builders

The utility of the expressions defined above is depicted in expression builders section of the Haskell code. For example, a boolean to be always true or false can be defined as:

```
true = B True
false = B False
```

Similarly, the equality operation can be defined as:

```
eq = Bin eqName
ge = Bin geName -- Greater or equal
```

Set operations can be built as:

```
setof = Agg setOpenName setCloseName
setnull = setof []
setdiff = Bin setdiffName
```

8.2.4 Abstract Syntax of Simple*Circus*

A concrete mathematical syntax for parameterised *CSP*, based on CSP_M [For05] extended with assignment, is:

$$
\begin{array}{rcl}
P,Q,R \in Proc & ::= & Div \mid STOP \mid SKIP \mid CHAOS_A \\
& \mid & a \to P \mid c?x?y:A!z \to P \mid P \mathbin{;} Q \\
& \mid & P \setminus A \mid P \Box Q \mid P \sqcap Q \mid c \,\&\, P \\
& \mid & P \mid\mid\mid Q \mid P \parallel_A Q \mid P\ {}_A\|_B\ Q \\
& \mid & {;}_{x:\sigma} \bullet P \mid \Box_{x:A} \bullet P \mid \bigsqcap_{x:A} \bullet P \\
& \mid & \mid\mid\mid_{x:A} \bullet P \mid \|^{A'}_{x:A} \bullet P \mid \|_{x:A} \bullet [A']P \\
& \mid & P \lhd b \rhd Q \mid N \mid N(e,f,\ldots) \mid x,y,\ldots := e,f,\ldots
\end{array}
$$

$$
\begin{array}{rcl}
D \in PDef & ::= & N \widehat{=} P \\
& \mid & N(x,y,\ldots) \widehat{=} P
\end{array}
$$

The implementation of the abstract syntax in our prototype is given below. The interesting bit in the syntax is type definition of `CircusProgram` and `CircusDef`. `CircusProgram` is a list of *Circus* definitions. Here each *Circus* definition or `CircusDef` consists of a name argument which identifies a process; followed by the list of `string` in which each element of the list depicts formal arguments i.e. the state variables of the process (in the *CSP* world, these state variables are going to turn into a parameter list); and body of the process which gives the description of the process through *Circus* actions which can be any of the actions in the *Circus* world available in the abstract syntax.

```
data Circus
 = Div | Stop | Skip |
 Chaos Expr |
 String :-> Circus |
 (String,[String]) ::-> Circus |
 Circus ::: Circus |
 IntChoice Circus Circus |
 ExtChoice Circus Circus |
 Hide Circus [String] |
 IPar [String] Circus Circus |
 APar [String] [String] Circus Circus |
 Ilv Circus Circus |
 Cond Expr Circus Circus |
 Guard Expr Circus |
 INT Name [String] Circus |
 SEQ Name [String] Circus |
 EXT Name [String] Circus |
 ILV Name [String] Circus |
 IPAR [String] Name [String] Circus |
 APAR Name [String] [String] Circus |
 Name := Expr |
```

```
 Call String [Expr]   --  name(actual arguments,)
 deriving (Eq,Ord,Show)

type CircusDef  --  name(formal arguments,) = body
   = ( String     -- name
     , ( [String]   -- formal argument
       , Circus     -- body
       )
     )

type CircusProgram = [CircusDef] -- ordered by name component
```

8.2.5 Precedence and Pretty Printing

A standard and compatible precedence is given to the logical, comparison and arithmetic operators. Also a precedence consistent with CSP_M constructs is provided. A complete list is in the appendix source code. Examples of this in Haskell are:

```
stdPrec nm
  | nm == eqvName = 21
  | nm == impName = 22
  | nm == orName = 23
  | nm == andName = 24
  | nm == eqName = 25
  | nm == neqName = 26
  | nm == ltName = 26
  | nm == leName = 26
  | otherwise = 1

cspPrec nm
  | nm == ilvName = 2
  | nm == interfaceParOpenName = 3
  | nm == alphabetisedParMidName = 3
  | nm == icName = 4
  | nm == ecName = 5
  | otherwise = 1
```

Pretty printing of expressions is provided by the function `ppExpr` which takes as input a precedence lookup function and expression and gives back a string. `Precf` takes as input a name and returns back the precedence. Pretty printing of some simple expressions is shown below:

```
type Precf = Name -> Int

ppExpr :: Precf -> Expr -> String

ppExpr prec e
  = pp 0 e
  where
   pp _ (B False) = "ff"
   pp _ (B True) = "tt"
```

```
pp _ (Z z) | z < 0 = brkt $ show z | otherwise = show z
pp _ (Var v) = v
```

Similarly, the `ppProc` function provides pretty printing for *Circus* processes. It takes the input a precedence and a *Circus* process and returns back a string. The pretty printing is available for all Simple*Circus* constructs, given in the abstract syntax of the language. As an example, the pretty printing implementation of some basic constructs and an alphabetised parallel is shown below:

```
ppProc :: Precf -> Circus -> String
ppProc prec e
  = pp 0 e
  where
    pp _ Div = "DIV"
    pp _ Stop = "STOP"
    pp _ Skip = "SKIP"

    pp cp (APar a b c1 c2)  -- p [a || a'] q
      = precRender prec cp alphabetisedParOpenName
        (\nmp -> pp nmp c1
             ++ ' ':alphabetisedParOpenName
++ ' ':(ppSts1 a)
++ ' ':alphabetisedParMidName
++ ' ':(ppSts1 b)
++ ' ':alphabetisedParCloseName
++ ' ':pp nmp c2)
```

8.2.6 *CSP* Laws

Algebraic laws govern the translation between the two languages. So, we formalised a significant number of the *CSP* laws available in [Ros98] using Haskell. The Haskell code included in Appendix lists all of them. There are forty four (44) *CSP* laws that are implemented in our prototype.

The approach here is that when a particular law is attempted on a *Circus* process, the first argument of the function sets a flag (a boolean) indicating if the law is applicable. If the boolean returns True, it means that the law was successfully applied to the process. Otherwise, it returns value of False in the first argument and the *Circus* process is returned unchanged. Two examples of the implementation of these laws are given below:

"; -assoc" $P;\ (Q;\ R) = (P;\ Q);\ R$

```
law_Seq_Assoc_LtoR (c1 ::: (c2 ::: c3)) = (True, ((c1 ::: c2) ::: c3))
law_Seq_Assoc_LtoR c = (False, c)

law_Seq_Assoc_RtoL ((c1 ::: c2) ::: c3) = (True, (c1 ::: (c2 ::: c3)))
law_Seq_Assoc_RtoL c = (False, c)
```

"\-||-dist" $(P\,{}_X\|_Y Q) \setminus Z = (P \setminus Z \cap X)_X\|_Y(Q \setminus Z \cap Y)$

```
law_Hide_APar_DistL (Hide (APar x y c1 c2) z)
    = (True, APar x y (Hide c1 zintsctx) (Hide c2 zintscty))
      where
         zintsctx = z `intersect` x
         zintscty = z `intersect` y

law_Hide_APar_DistL c = (False, c)

law_Hide_APar_DistR z x' y' z' circ@(APar x y (Hide c1 zintsctx')
                                              (Hide c2 z'intscty'))
    | x==x' && y==y' && z==z' = (True, Hide (APar x y c1 c2) z)
    | otherwise = (False, circ)
      where
         zintsctx' = z `intersect` x'
         z'intscty' = z' `intersect` y'
```

8.2.7 Head Normal Form Implementation in Haskell

We have explained the head normal forms (HNF) of *CSP* in Section 2.6, on page 16. The normal form implementation in Haskell is given here. The basic constructs are already in head normal form.

```
hnf Stop = Stop
hnf Skip = Skip
hnf Div = Div
hnf circ@((x,xs)::->c) = circ
```

For internal choice required to be in HNF,
$hnf(Q \sqcap R) \mathrel{\widehat{=}} hnf(Q) \sqcap hnf(R)$

```
hnf (IntChoice c1 c2) = IntChoice (hnf c1) (hnf c2)
```

For interleaving to be transformed to HNF involves the application of the distributive law to the head normal form of the left and right processes of the interleaving. Then, the application of the step law for ||| turns the whole expression into head normal form.

"|||-step"

$$\begin{array}{ll} x : C \to P(x) \; ||| \; y : D \to Q(y) & \text{"|||-step"} \\ \quad = \quad x : C \to (P(x) \; ||| \; y : D \to Q(y)) & \\ \quad \sqcap \quad y : D \to (x : C \to P(x) \; ||| \; Q(y)) & \end{array}$$

```
hnf (Ilv c1 c2) = (snd $ law_ILeave_StepL
                     (Ilv (snd $ law_Ilv_DistL (hnf c1)) (snd $ law_Ilv_DistL (hnf c2))))
```

The same strategy is to adopt for acquiring HNF of the hiding, external choice, internal choice, interface parallel, and alphabetised parallel constructs. Here, we represent the step law of hiding construct. Meanwhile, the remaining are included in the appendix C.6, on page 157.

“\-step”

$$(a \rightarrow P) \setminus H = \begin{cases} a \rightarrow (P \setminus H), & \text{if } a \notin H \\ P \setminus H, & \text{if } a \in H \end{cases}$$

A generalised form of this law is:

$$(a : A \rightarrow P) \setminus H = (a : (A \setminus H) \rightarrow (P \setminus H)) \sqcap \bigsqcap_{a : A \cap H} \bullet (P \setminus H)$$

```
hnf (Hide circ a) = (snd $ law_Hide_StepL (Hide (snd $ law_Hide_DistL (hnf circ)) a))
```

8.2.8 Implementation of the Step Laws

Step laws are integral part of the process of acquiring the head normal form for a *Circus* process.

Normalisation refers to a process of changing sides of an equation to obtain a required form. As a process algebra, *CSP* processes can be normalised to a particular shape to get better performance using the model checker of the language, i.e. FDR [For05]. This normalisation process involves getting the head normal form, abbreviated as HNF. This process is explained in Chapter 11, [Ros98]. We implemented this in Haskell. While some basic constructs and equations are already in normal form, the normal form is the internal choice of external choices of guarded prefix actions, leading to other normal form expressions. Here, we just present the normalisation process of the external choice. In general, it involves the distributive laws which remove the non-deterministic or internal choices to the right; the step laws which calculate the first-step actions of constructs; and the manipulations of the expressions within the processes. For converting external choice required in HNF, involves the application of the distributive law to the head normal form of the left and right processes of the external choice. Then, the application of the step law turns the whole expression into HNF.

```
hnf (ExtChoice c1 c2) = (snd $ law_ExtChoice_StepL
      (ExtChoice (snd $ law_ExtChoice_IntChoice_Dist_LtoR (hnf c1))
      (snd $ law_ExtChoice_IntChoice_Dist_LtoR (hnf c2))))
```

Here the step law for external choice shown in the figure 8.2.8 below implemented in Haskell as:

“□-step”

$$\begin{array}{ll} x : A \rightarrow P(x) \,\Box\, y : B \rightarrow Q(y) & \text{“□-step”} \\ \quad = \;\; x : (A \setminus B) \rightarrow P(x) & \\ \quad \Box \;\; z : (A \cap B) \rightarrow (P(z) \sqcap Q(z)) & \\ \quad \Box \;\; y : (B \setminus A) \rightarrow Q(y) & \end{array}$$

Figure 8.9: The Step Law for External Choice, □

```
law_ExtChoice_StepL circ@(ExtChoice ((x,xs) ::-> c1) ((y,ys) ::->
   c2)) = (True, mkExtChoice p1 (mkExtChoice p2 p3))
   where
     p1 = (x,adiffb) ::-> c1
     p2 = (z,aintsctb) ::-> (mkIntChoice p4 p5)
     p3 = (y,bdiffa) ::-> c2
     adiffb = xs \\ ys
     aintsctb = xs `intersect` ys
     bdiffa = ys \\ xs
```

```
     p4 = (csubstitute x z c1)
     p5 = (csubstitute y z c2)
     z = freshName $ circNames circ
law_ExtChoiceStepL c = (False, c)
```

In addition, the implementation of the step law for $(X\|Y)$ in Haskell is also shown here.

```
law_APar_StepL circ@(APar a b ((x,xs)::->c1) ((y,ys)::->c2))
        = (True, mkExtChoice p1 (mkExtChoice p2 p3))
        where
           p1 = ((x,comb1) ::-> (APar a b c1 ((y,ys)::->c2)))
           p2 = ((z,comb2) ::-> (APar a b c3 c4))
           p3 = ((y,comb3) ::-> (APar a b c5 c2))
           comb1 = ((xs `intersect` a) \\ b)
           comb2 = ((xs `intersect` (a `intersect` (ys `intersect` b))))
           comb3 = ((ys `intersect` b) \\ a)
           c3 = (csubstitute x z c1)
           c4 = (csubstitute y z c2)
           c5 = ((x,xs)::->c1)
           z = freshName $ circNames circ
law_APar_StepL c = (False, c)
```

8.2.9 Top Level Translator Function Implementation

In the implementation, `action2csp` is a top level function which takes a complete *Circus* program and a defined action in it and returns its equivalent in the *CSP* world. The implementation is achieved by making use of `mkCGraph` appearing later in the implementation. After getting the graph of a particular *Circus* program through `mkCGraph`, the `translateCirc` function is used to do the actual translation based on the action name received and its corresponding variables and calls.

```
action2csp :: CircusProgram -> String -> CircusProgram
action2csp prog aname
 = let cgrf = mkCGraph prog
   in case alookup cgrf aname of
        Nothing  ->  error ("No action '"++aname++"' found")
        Just (vars,calls)  -> translateCirc prog cgrf aname vars calls
```

The difference between high level functions `circus2csp` and `action2csp` is that the function `circus2csp` maps a complete *Circus* program to its translated version while the latter works on an individual action inside the *Circus* program.

```
circus2csp :: CircusProgram -> CircusProgram
circus2csp prog
 = let cgrf = mkCGraph prog
   in prog
```

Function `getCircVarsCalls` takes a complete *Circus* program and gets the information involved, i.e. the definition name, the array maintaining variables used in the definition and the array having the information about the calls of particular actions. This is achieved by using functions `actionVars` and `actionCalls`.

```
getCircVarsCalls :: CircusProgram
            -> [ ( String  -- definition name
                 , ( [String] -- variables used in definition (sorted)
                   , [String] -- actions called
                   )
                 )
               ]
getCircVarsCalls defs
 = alnorm $ gCVC defs
 where
  gCVC [] = []
  gCVC  ((aname,(aparam,abody)):rest)
   = (aname,(avars,acalls)):gCVC rest
   where
    avars = lnorm $ actionVars abody
    acalls = lnorm $ actionCalls abody
```

The function `actionVars` takes a particular *Circus* action and generates an array for gathering names of the variables used in a particular action. Similarly, the function `actionCalls` takes a particular *Circus* action and generates an array for gathering names of the calls to particular actions.

To gather information on the variables used in a particular expression, the function `exprVars` is defined.

```
exprVars (Var v) = [v]
exprVars (Agg _ _ es) = concat $ map exprVars es
exprVars (App _ es) = concat $ map exprVars es
exprVars (Bin _ e1 e2) = exprVars e1 ++ exprVars e2
exprVars _ = []
```

The important function here is `detCircDeps` which manages the record of the dependencies for the final translated version of the *Circus* program. It determines the dependencies by analysing the calls to particular actions by using the `getCalls` function.

```
detCircDeps :: [(String,([String],[String]))] -> [(String,([String],[String]))]
detCircDeps deps
 = dCP deps [] False deps
 where

   dCP deps0 deps' chgd []
     | chgd  =  dCP deps' [] False deps'
     | otherwise  =  deps' -- should equal deps0 !

   dCP deps0 deps' chgd  (dep@(name,(vars,calls)):rest)
     | calls' == calls  =  dCP deps0 (dep:deps') chgd rest
     | otherwise  =  dCP deps0 ((name,(vars,calls')):deps') True rest
```

```
     where calls'  = getCalls deps0 calls calls

getCalls :: [(String,([String],[String]))] -> [String] -> [String] -> [String]
getCalls deps0 calls' [] = lnorm calls'
getCalls deps0 calls' (call:calls)
 = case alookup deps0 call of
     Nothing  ->  error ("Action '"++call++"' is undefined")
     Just (_,subcalls) ->  getCalls deps0 (subcalls++calls') calls
```

As mentioned earlier, the functions `mkCGraph` and `translateCirc` are used by the high level functions of translator called `action2csp` and `circus2csp`. After getting the graph of a particular *Circus* program through `mkCGraph`, the `translateCirc` function is used to do the actual translation based on the action name received and its corresponding variables and calls.

```
mkCGraph :: CircusProgram -> [(String,([String],[String]))]
mkCGraph = detCircDeps . getCircVarsCalls


translateCirc :: CircusProgram -> [(String,([String],[String]))]
                               -> String -> [String] -> [String] -> CircusProgram
translateCirc prog cgrf aname vars calls
 = let usedActionNames = lnorm (aname:calls)
       isUsed (nm,_) = nm `elem` usedActionNames
       rprog = filter  isUsed prog

       newplist = lnorm (vars ++ getParams cgrf calls)
       plistvars = map Var newplist
       addpars (nm,(pars,body))
        = (nm,(pars++newplist,addParams nm plistvars body))

       pprog = map addpars rprog

   in pprog
```

The purpose of `addParams` function is to attach the list of parameters to a *Circus* action. This is required because in the *CSP* world the variables of the *Circus* world turn into parameters and a particular action is called using parametric calls.

```
addParams :: String -> [Expr] -> Circus -> Circus
addParams nm plist (Call cnm pars) = (Call cnm (pars++plist))
```

The function is also important, when `Skip` construct is visited. Whenever, a `Skip` appears, it is replaced with the continuation marker. Later on, in the translation, these continuation markers are replaced with the calls, already obtained from the analysis of the main action.

```
addParams nm plist Skip = (Call (nm++"_CONT") plist)
```

When all other constructs available in the syntax tree are visited, they recursively call `addParams` function.

```
addParams nm plist (c1 ::: c2) = (addParams nm plist c1 ::: addParams nm plist c2)
addParams nm plist (a :-> circ) = (a :-> (addParams nm plist circ))
addParams nm plist ((x,xs) ::-> circ) = ((x,xs) ::-> addParams nm plist circ)
```

```
addParams nm plist (IntChoice c1 c2) = (IntChoice (addParams nm plist c1)
                                                  (addParams nm plist c2))
addParams nm plist (ExtChoice c1 c2) = (ExtChoice (addParams nm plist c1)
                                                  (addParams nm plist c2))
```

When no pattern is matched, the function `addParams` returns the action unchanged.

```
addParams nm plist body = body
```

The function `getParams` is there to extend the parameter list, if new parameters are added in by a particular action.

```
-- extract part of prog of interest - all definitions aname:calls (rprog)
-- add newplist to extend plist of every call (pprog)
-- return pprog

getParams :: [(String,([String],[String]))] -> [String] -> [String]
getParams cgrf calls = concat $ map (fst . fromJust . alookup cgrf)  calls
```

The function `extractSeq` is defined here to generate fresh names for the *Circus* actions on the left and right side of the sequential composition. Each leading node from the tree of sequential compositions is given a fresh name to make the *Circus* action compositions a sequential one. For example:
$(A \Box B)$; $(C \Box D) => N_1; N_2$ where $N_1 = (A \Box B)$ and $N_2 = (C \Box D)$.

```
getExtractSeq :: CircusProgram -> [ (Circus, [(String, Circus)]) ]
getExtractSeq defs
   = gES defs
 where
  gES [] = []
  gES ((aname,(aparam,abody)):rest)
   | aname == "MAIN" = (extractSeqSt):gES rest
   | otherwise = gES rest
   where
     extractSeqSt = extractSeq abody

extractSeq :: Circus -> (Circus, [(String,Circus)])
extractSeq circ = (circ, newNamedActs circ)

newNamedActs :: Circus -> [(String, Circus)]

newNamedActs (ExtChoice c1 c2) =
               [(newActNameGen c1, c1)] ++ [(newActNameGen c2, c2)]
newNamedActs (IntChoice c1 c2) =
               [(newActNameGen c1, c1)] ++ [(newActNameGen c2, c2)]
newNamedActs (c1 ::: c2) = (newNamedActs c1) ++ (newNamedActs c2)
newNamedActs _ = []
newActNameGen :: Circus -> String
newActNameGen circ = (freshNameAct $ circNames circ)
```

8.2.10 Implementation of Formalised Steps in Translation Theory

The translation process steps formalised in Chapter 7 are implemented in Haskell code as well. Section C.9 on page 169 includes this implementation. The top level translator functions at action and process level are `topTranslatorAct` and `topTranslator`. The process flow is depicted in figure 8.10.

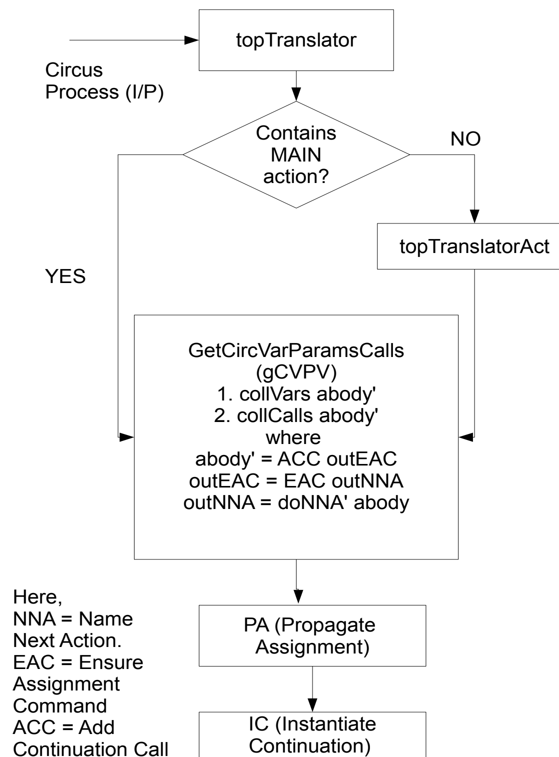


Figure 8.10: Top Level Translator Working Flow based on Formalised Steps

When a *Circus* program is given as input to the `topTranslator` function, first, it checks that the program contains a main action. If a main action is there, the `getCircVarParamsCalls` function is called. It collects state variables and the action calls (final transformation will be based on call dependencies in the main action). The action body `abody` will be updated to `abody'` by following the formalised steps defined as functions. For example: the `doNNA` function performs the "Name Next Action" step on the action body; the `doEAC` is for the "Ensure Assignment Continuation" step; the `doACC` is for the "Add Continuation Call" step; the `doPA` is for "Propagate Assignments"; and the `doIC` is for the "Instantiate Continuations" function. All of these steps are mathematically represented in Section 7.2, on page 58. If a particular *Circus* program does not contain the main action, it decides to use the `topTranslatorAct` function instead.

Here, we include the implementation of the "Ensure Assignment Continuation" step implementation in Haskell to show the correspondence between the mathematical representation, actual functionality and the implementation detail of this step.

Ensure Assignment Command: Every assignment $x := e$ that is not immediately followed by $\fatsemi$ we replace

with $x := e \mathbin{;} Skip$.

$$
\begin{array}{rcl}
EAC & : & Var^* \to Action \to Action \times ActDef \\
EAC_V(x := e \mathbin{;} B) & \widehat{=} & x := e \mathbin{;} EAC_V(B) \\
EAC_V(A \mathbin{;} B) & \widehat{=} & EAC_V(A) \mathbin{;} EAC_V(B) \\
EAC_V(x := e) & \widehat{=} & x := e \mathbin{;} Skip \\
EAC_V(F\,A) & \widehat{=} & F\,(EAC_V(A)) \\
EAC_V(A \oplus B) & \widehat{=} & EAC_V(A) \oplus EAC_V(B) \\
EAC_V(A) & \widehat{=} & A \quad \text{—when none of the above apply...}
\end{array}
$$

The function `ensureAssgnCont` checks all actions in a *Circus* program for assignment commands, not followed by *Skip*. If such assignment command is found in the action definition, a *Skip* is sequentially composed to the assignment command.

```
ensureAssgnCont :: Circus -> Circus

ensureAssgnCont circ@(asg@(n := e) ::: c) = asg ::: ensureAssgnCont c

ensureAssgnCont (n := e) = (n := e) ::: Skip
```

When all other constructs available in the syntax tree are visited, they recursively call `ensureAssgnCont` function.

```
ensureAssgnCont (c1 ::: c2) = ensureAssgnCont c1 ::: ensureAssgnCont c2
ensureAssgnCont ((x,xs) :-> circ) = ((x,xs) :-> ensureAssgnCont circ)
```

8.3 Summary

This chapter started with the implementation details for the circus2cspm tool. We achieved the correct output of a number of translation examples with a subset of constructs from *Circus* to CSP_M. However, we found the development in Java quite complex especially in the record keeping process and moving from initial to intermediate and then from intermediate to the final transformation of the translated version. We dealt here with some simple action compositions. But handling more of the various language construct cases proved very hard to implement. Pattern matching for the compositions of complex combinations of actions in the main action is hard to find and implement in the present state of the source code of circus2cspm. This pattern matching is also a pre-requisite while dealing with the implementation of all *CSP* laws in the main action. This goal was found hard to achieve by implementation in Java.

We then presented the implementation details of the SimpleCircus2CSPM prototype. This is a novel approach of prototyping the translation from *Circus* to CSP_M, exploiting the expressiveness of functional languages. In the Haskell implementation, a number of examples are provided. We covered a number of constructs in our prototype such as prefix actions, sequential composition, internal choice, external choice, hiding action, interface parallel, alphabetised parallel, conditional, assignment command, calling an action, as well as their generalised forms. A significant number (forty four) of CSP algebraic laws are formalised in the implementation. The normalisation process is also implemented in the prototype. A simple implementation of pretty printing is also provided which can be extended to provide a more sophisticated one. The formalised steps of translation, defined in Chapter 7, are implemented in Haskell code. The next chapter evaluates prototype tools we developed, by inspecting the running examples and their corresponding outputs.

Chapter 9

Evaluation

This chapter evaluates the translated examples from the SimpleCircus2CSPM tool and gives comments on the results achieved so far. It also describes the case study of a cache coherence protocol specified in the notation of Simple*Circus* in order to explain its utility for specifying such protocols. Subsequently, the *Circus* examples translated through the tool circus2cspm are discussed.

9.1 Running Examples in SimpleCircus2CSPM

9.1.1 Example 1 – Simple Sequential Case in the Main Action

This example contains three actions, A, B, and C. There are two variables, x and y defined in these actions. Meanwhile, the third variable named z is introduced in the main action. The actions are made up of assignment commands, sequential composition, prefix action and one of the basic actions, that is, *Skip*. The main action defined is kept simple with just sequential compositions between each call of a particular action.

The mathematical version of the *Circus* example, $cp1$, is:

$$A \mathrel{\widehat=} (x := 1) \mathbin{;} (a \rightarrow Skip)$$
$$B \mathrel{\widehat=} (y := x + 1) \mathbin{;} (b \rightarrow Skip)$$
$$C \mathrel{\widehat=} (c \rightarrow Skip)$$
$$MAIN \mathrel{\widehat=} (z := 0) \mathbin{;} A \mathbin{;} B \mathbin{;} C$$

The output from translator is as follows, using the command, `putppd (circus2csp cp1)`:

```
A(x,y,z) =^= a -> B(1,y,z)

B(x,y,z) =^= b -> C(x,x+1,z)

C(x,y,z) =^= c -> Skip

MAIN(x,y,z) =^= A(x,y,0)
```

Comments: By inspecting the mathematical version of this example, it can be seen that the main action of the process calls actions A, B, and C in sequence. Before calls to actions, the assignment command sets the value of variable z equal to zero. The output from `circus2csp` function shows that the process starts by calling action A, while the expression of z is substituted in the parameter list. Afterwards, after performing event a, action B is called with the parameter list having being updated by the assignment command of setting variable x value to 1. Similarly, action B calls actions C with parameter list having being updated by the assignment

command of setting variable y value to $x+1$. As action C is the last in the sequential composition of calls in the MAIN so we find that after performing event c, the process terminates. The correct output should contain the observable events as $a \rightarrow b \rightarrow c \rightarrow Skip$ which is achieved in the translated version.

9.1.2 Example 2 – Adding Extra Disjoint Variables in the Main Action

The mathematical version of the *Circus* example $cp2$ is:

$$A \mathrel{\widehat=} (x := 1) \mathbin{⨾} (a \rightarrow Skip)$$
$$B \mathrel{\widehat=} (y := x+1) \mathbin{⨾} (b \rightarrow Skip)$$
$$C \mathrel{\widehat=} (c \rightarrow Skip)$$
$$MAIN(p,q) = (z := 0) \mathbin{⨾} A \mathbin{⨾} B \mathbin{⨾} C$$

Comments: This *Circus* process is same as $cp1$. The difference here is the inclusion of extra variables (p,q) by the main action. It can be seen that these variables are disjoint from the ones in the actions A, B, and C. So, the call to A from main just includes parameters x, y and z.

The output from translator function `circus2csp` is as follows, using the command, `putppd (circus2csp cp2)`:

```
A(x,y,z) =^= a -> B(1,y,z)

B(x,y,z) =^= b -> C(x,x+1,z)

C(x,y,z) =^= c -> Skip

MAIN(P,Q,x,y,z) =^= A(x,y,0)
```

Comments: The correct output should contain the observable events as $a \rightarrow b \rightarrow c \rightarrow Skip$ which is achieved in the translated version. Also the subsituted expression in each call to a particular action is achieved correctly. The disjoint variables from the main action are not included in call to action A, which is as required in the correct translation.

9.1.3 Example 3 – *Circus* Process without the Main Action

The mathematical version of the *Circus* example $cp3$ is:

$$A \mathrel{\widehat=} (x := 1) \mathbin{⨾} (a \rightarrow B)$$
$$B \mathrel{\widehat=} (y := x+1) \mathbin{⨾} (b \rightarrow C)$$
$$C \mathrel{\widehat=} (c \rightarrow A)$$

Comments: Here, in this example, there is no MAIN action. If this example is given as an input to `circus2csp` function, the translator chooses to use the action level translation instead. The action level translator function `action2csp` determines the dependencies for each action and translates the *Circus* process.

The output from the translator is as follows, using the command `putppd (action2csp cp3 "A")`

```
A(x,y) =^= a -> B(1,y)

B(x,y) =^= b -> C(x,x+1)

C(x,y) =^= c -> A(x,y)
```

Comments: Here, we can see that the function `action2csp` is producing the right *CSP* output, as it determines the dependencies for the whole process. Also the subsituted expression in each call to a particular action is achieved correctly.

9.1.4 Example 4 – Two Distinct Sequential Composition Chains of Calling Actions

In the *Circus* example *cp*4, there is no main action. Here, actions *A*, *B*, and *C* are calling each other. Action *D* calls itself, while action *E* calls action *D*.

The mathematical version of the *Circus* process *cp*4 is:

$$A \widehat{=} (x := 1) \mathbin{;} (a \rightarrow B)$$
$$B \widehat{=} (y := x + 1) \mathbin{;} (b \rightarrow C)$$
$$C \widehat{=} (c \rightarrow A)$$
$$D \widehat{=} (a := b + 1) \mathbin{;} (d \rightarrow D)$$
$$E \widehat{=} (e \rightarrow D)$$

As there is no main action, so we have to use the `action2csp` function, instead of the `circus2csp` funtion. The output from the translator for action *A* is as follows, using the command `putppd (action2csp cp4 "A")`:

```
A(x,y) =^= a -> B(1,y)

B(x,y) =^= b -> C(x,x+1)

C(x,y) =^= c -> A(x,y)
```

The output from translator for action *D* is as follows, using the command `putppd (action2csp cp4 "D")`:

```
D(a,b) =^= d -> D(b+1,b)
```

The output from translator for action *E* is as follows, using the command `putppd (action2csp cp4 "E")`:

```
D(a,b) =^= d -> D(b+1,b)

E(a,b) =^= e -> D(a,b)
```

Comments: Here again, we can see that the translator function `action2csp` is producing the right *CSP* output for the individual actions in the whole process definition, as it determines the dependencies correctly in case of the actions *A*, *D* and *E*. The translator correctly identifies that *A*, *B* and *C* use different variables from *D* and *E*.

9.1.5 Example 5 – the Lift Process

This example is more elaborate as it involves more expression builders as well as guarded commands with external choice between them. The main action calls the initialiser which identifies the variables for the floor number and the state of the lift door. Afterwards, the Lift process operates on the basis of the guarded commands and performs events of up, down, close and open. Up and down events change the floor number while close and open events change the status of the lift door.

The mathematical version of the lift example is:

$$
\begin{array}{l}
INITLIFT \mathrel{\widehat=} (floor := 1) \mathbin{;} (doorState := closed) \\
LIFT \mathrel{\widehat=} (floor < 5 \land doorState = closed \& (up \rightarrow floor := floor + 1 \mathbin{;} LIFT)) \\
\Box \\
(floor > 0 \land doorState = closed \& (down \rightarrow floor := floor - 1 \mathbin{;} LIFT)) \\
\Box \\
(doorState = open \& (close \rightarrow doorState := closed \mathbin{;} LIFT)) \\
\Box \\
(doorState = closed \& (open \rightarrow doorState := opened \mathbin{;} LIFT)) \\
MAIN = INITLIFT; LIFT
\end{array}
$$

The output from translator function `circus2csp` is as follows, using command, `putppd (circus2csp cp5)`:

```
INITLIFT(doorState,floor) =^= LIFT(closed,1)

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

MAIN(doorState,floor) =^= INITLIFT(doorState,floor)
```

Comment: Here, the translator is producing the correct output. The main action calls the initialiser function `INITLIFT`. Here, the initial status of `LIFT` is set by passing the initial values of the parameters. The `doorState` parameter is set to `closed`, while floor number is set to 1 initially. Then, depending on the events *up*, *down*, *close*, and *open*, the `LIFT` process continues by calling itself recursively.

9.1.6 Example 6 – Including External Choice in the Main Action

The *Circus* example *cp*7 contains an external choice between two disjoint sequentially composed chains of calls. Here, the left hand side sequentially composed chain has calls to actions A, B, and C. Meanwhile, the right hand side chain has calls to action D and E.

$$
\begin{array}{l}
A \mathrel{\widehat=} (x := 1) \mathbin{;} (a \rightarrow B) \\
B \mathrel{\widehat=} (y := x + 1) \mathbin{;} (b \rightarrow C) \\
C \mathrel{\widehat=} (c \rightarrow A) \\
D \mathrel{\widehat=} (u := v + 1) \mathbin{;} (d \rightarrow E) \\
E \mathrel{\widehat=} (e \rightarrow D) \\
MAIN \mathrel{\widehat=} (A \mathbin{;} B \mathbin{;} C) \Box (D \mathbin{;} E)
\end{array}
$$

The output from translator is as follows, using the command `putppd (circus2csp cp7)`:

```
A(u,v,x,y) =^= a -> B(u,v,1,y)

B(u,v,x,y) =^= b -> C(u,v,x,x+1)

C(u,v,x,y) =^= c -> A(u,v,x,y)

D(u,v,x,y) =^= d -> E(v+1,v,x,y)

E(u,v,x,y) =^= e -> D(u,v,x,y)

MAIN(u,v,x,y) =^= A(u,v,x,y)
           []
         D(u,v,x,y)
```

Comments on Output: Here, in this case, the translator function `circus2csp` produces the correct output. The first thing to observe is that the state variables defined at all the places in *Circus* process are collected successfully. Secondly, the expression substitution for all the assignment commands is working fine. The left hand side of the main action calls the left sequentially composed chain by just calling action A. On the other hand, the right hand side of the main action calls the right sequentially composed chain by just calling action D.

9.2 Case Study – A Cache Coherence Protocol Representation in Simple*Circus* Notation

The work presented in [FS96] is selected to be represented in the Simple*Circus* notation and implemented in Haskell.

9.2.1 Background Information

Caches are fast memories placed in computer systems in order to have fast access to the items being frequently used by the processors during program execution. If multiple processes are accessing same data in the cache, the program should pick the most recently updated value from the cache. The presence of either multiprocessors or multiple caches introduces the problem of coherence. To deal with this problem, cache coherence protocols are being implemented in memory systems.

The authors of [FS96] modelled and proved correctness for the cache coherence part of the Scalable Coherent Interface (SCI) – an IEEE standard for specifying communication among processors. This was done in a *guarded command language*. For more technical details, the work in [FS96] is the reference.

The protocol under study was modelled as processes for memory (m) and processor (p). There were seventeen (17) processes at processor level and four (4) processes at memory level. Each process describes its status and a finite set of guarded actions.

The specification of all of these processes is done here using the notation of Simple*Circus*. Here, we include two very simple processes on the processor side. Then, we include two processes on the memory side. The remaining processes are included in the appendix D, on page 179.

9.2.2 The Felty Example in Haskell

We import Haskell modules developed for SimpleCircus2CSPM to specify the cache coherence protocol specified in [FS96] using Simple*Circus* notation.

```
module Felty where
import Data.List
import Maybe
import Utilities
import StdSimpleCircus
```

The state variables in the Felty model are mapped to the following abbreviations in our model:

For *h*, we use "hist".
For $status_m$, we use "statusMem".
For $buf[m]$, we use "buf[m]".
For $head_m$, we use "headMem".
For *nil*, we use "Nil".
For $status_p$, we use "statusProc".
For $buf[p]$, we use "buf[p]".
For cs_p, we use "csProc".
For *invalid*, we use "invalid".
For $succ_p$, we use "succProc".
For $pred_p$, we use "predProc".

The initial conditions of the model are as below:

- $h = \varepsilon$
- $status_m = Home \wedge buf[m] = \varepsilon \wedge head_m = nil$, and
- for all p $\in$ P, $status_p = Off \wedge buf[p] = \varepsilon \wedge$ $cs_p = invalid \wedge succ_p = nil \wedge pred_p = nil$.

```
initcp6 = [ ("INIT_SCI_MODEL", ([], ("hist" := setnull) :::
                                    ("statusMem" := (Val "Home")) :::
                                    ("bufMem" := setnull) :::
                                    ("headMem" := (Val "Nil")) :::
                                    ("statusProc" := (Val "Off")) :::
                                    ("bufProc" := setnull) :::
                                    ("csProc" := (Val "invalid")) :::
                                    ("succProc" := (Val "Nil")) :::
                                    ("predProc" := (Val "Nil"))
                               )
              )
          ]
```

The auxiliary functions given below are defined to simplify the specification of memory and processor processes in our model.

```
bufout i act = Evts ("buf["++i++"]",[Bang (Val act)])
bufin i act = Evts ("buf["++i++"]",[Q act Nothing])

read_cache_freshQ p = "read_cache_freshQ("++p++")"
read_cache_goneQ p = "read_cache_goneQ("++p++")"
```

```
read_cache_freshR p = "read_cache_freshR("++p++")"
read_cache_goneR p = "read_cache_goneR("++p++")"

prependQ p = "prependQ("++p++")"
prependR p = "prependR("++p++")"

purgeQ p = "purgeQ("++p++")"
purgeR p = "purgeR("++p++")"

modifydataQ p = "modifydataQ("++p++")"
modifydataR p = "modifydataR("++p++")"

delrightQ p = "delrightQ("++p++")"
delrightR p = "delrightR("++p++")"

delleftQ p = "delleftQ("++p++")"
delleftR p = "delleftR("++p++")"
```

On page 3 of [FS96], the description of processes $p1$ and $p2$ are as follows:

If processor p is in the *Off* state then it can send a message named *read_cache_freshQ(p)* to memory indicating that p wants to read the cache; or a message *read_cache_goneQ(p)* indicating that p wants to modify the cache. Processor p then goes to the *Pending* state waiting for a response from memory.

Formal Description of process $p1$ and $p2$:

($p1$) $status_p = Off \rightarrow$

$buf[m]!read_cache_freshQ(p); status_p{:=}Pending$

($p2$) $status_p = Off \rightarrow$

$buf[m]!read_cache_goneQ(p); status_p{:=}Pending$

```
cp6p1 = [
        ( "P1"
          , ( []
            , Guard (eq (Var "statusProc") (Val "Off"))
                    ( bufout "m" (read_cache_freshQ "p")
                     :::->
                    ("statusProc" := (Val "Pending")))
            )
          )
        ]
cp6p2 = [
        ( "P2"
          ,  ( []
             , Guard (eq (Var "statusProc") (Val "Off"))
                      ( bufout "m" (read_cache_goneQ "p")
                       :::->
                      ("statusProc" := (Val "Pending")))
             )
          )
        ]
```

On page 3 of [FS96], the description of process $m1$ is as follows:

If memory m receives *read_cache_freshQ(p)* then it sends a message *read_cache_freshR* as a response to p. This message carries 4 arguments.
i. identity of m
ii. the processor which will be p's successor in the shared list (this will be nil if there is no other processor in the queue).
iii. value of cv_m
iv. either *gone* if m is not the owner of the cache or *ok* otherwise.
Memory also updates its variable $head_m$. If p is the first processor on the list from m's point of view, then m goes from the *Home* state to *Fresh* state. This is given as the $m1$ process.
Formal Description of process m1:

($m1$) *buf[m]*?*read_cache_freshQ(p)* $\rightarrow$
if $status_m$ = *Gone*
then *buf[p]*!*read_cache_freshR(m,*$head_m$,cv_m*,gone)*
else *buf[p]*!*read_cache_freshR(m,*$head_m$,cv_m*,ok)*
fi; $head_m$:=p;
if $status_m$ = *Home* **then** $status_m$:= *Fresh* **fi**

```
cp6m1 = [
           ("M1"
          , ([]
            , (bufin "m" (read_cache_freshQ "p")
              :::->
              (Cond (eq (Var "statusMem") (Val "Gone"))
                 (bufout "p" (read_cache_freshR "m,headMem,cvMem,gone")
                    :::->Skip)
                 (bufout "p" (read_cache_freshR "m,headMem,cvMem,ok")
                    :::->Skip)
              )
              ::: ("headMem" := (Val "p"))
              ::: (Cond (eq (Var "statusMem") (Val "Home"))
                        ("statusMem" := (Val "Fresh")) Skip)
              )
            )
          )
         ]
```

On page 3 of [FS96], the description of process $m2$ is as follows:

If memory m receives message *read_cache_goneQ(p)*, then it sends a message *read_cache_goneR* back to p. This message also carries 4 arguments as mentioned above for the case of *read_cache_freshQ(p)*. As previous case, m updates its variable $head_m$. Finally, m goes to the *Gone* state. This is given as the $m2$ process.
Formal Description of process m2:

($m2$) *buf[m]*?*read_cache_goneQ(p)* $\rightarrow$
if $status_m$ = *Gone*
then *buf[p]*!*read_cache_goneR(m,*$head_m$,cv_m*,gone)*
else *buf[p]*!*read_cache_goneR(m,*$head_m$,cv_m*,ok)*
fi; $head_m$:=p; $status_m$:= *Gone*

```
cp6m2 = [
      ("M2",
             ([]
             , (bufin "m" (read_cache_goneQ "p")
                :::->
                (Cond (eq (Var "statusMem") (Val "Gone"))
                 (bufout "p" (read_cache_goneR "m,headMem,cvMem,gone") :::-> Skip)
                 (bufout "p" (read_cache_goneR "m,headMem,cvMem,ok") :::-> Skip)
                )
               ::: ("headMem" := (Val "p"))
               ::: ("statusMem" := (Val "Gone"))
               )
             )
          )
        ]
```

As mentioned earlier, the remaining processes are included in appendix D, on page 179.

9.3 Running Examples in circus2cspm Tool

Here, we describe again how the circus2cspm tool works. The interface of circus2cspm is quite simple. It takes as input a LaTeX file. Then the user specifies the output file directory where the output project directory structure is to be created. If correctly parsed and type checked, the CSP_M output is generated. The datatype declaration is in the output file called `DataType.csp` where `DataType` is replaced with the name of the actual data type given in the *Circus* specification. Channel declarations are listed in `ChannelDecl.csp`, while the `finalProcDecl.csp` is the file containing the final translated version of the process definition.

9.3.1 Example 1 – Lift Process

This example is same as the one translated by the SimpleCircus2CSPM prototype, given on page 87.

$DoorState ::= opened \mid closed$

channel $up, down, open, close$
process $Lift \mathrel{\widehat=}$ **begin**
state

LiftState

$floor : \mathbb{N}$
$doorState : DoorState$

$InitLift \mathrel{\widehat=} (floor := 0;\ doorState := closed)$

$Lift \mathrel{\widehat=} (floor < 5 \land doorState = closed \,\&\, up \rightarrow floor := floor + 1)$
$\quad \Box\ (floor > 0 \land doorState = closed \,\&\, down \rightarrow floor := floor - 1)$
$\quad \Box\ (doorState = closed \,\&\, open \rightarrow doorState := opened)$
$\quad \Box\ (doorState = opened \,\&\, close \rightarrow doorState := closed)$

$\bullet\ InitLift;\ \mu X \bullet (Lift;\ X)$

end

The CSP_M script generated from circus2cspm tool is as follows:
The datatype `DoorState` is defined in the generated file `DoorState.csp`, with the following script:

```
datatype DoorState = opened | closed
```

The channel declaration code is written to the file named `ChannelDecls.csp`, having the following script:

```
channel up
channel down
channel open
channel close
```

The final translated version is written to `ProcDecls.csp` file, having the following script.

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

9.3.2 Example 2 – Simple Sequential Chain of Calls in Initialiser

This example is the same as the one translated from the SimpleCircus2CSPM prototype, given on page 85. The difference here is the order of calling actions in the initialiser.

channel a, b, c

process $ProcEx1 \mathrel{\widehat=}$ **begin**

state

Ex1State

$x, y : \mathbb{N}$

$InitProcEx1 \mathrel{\widehat=} x := 1;\ y := 1$
$Action_A \mathrel{\widehat=} x := 0;\ a \rightarrow Skip$
$Action_B \mathrel{\widehat=} y := x + 1;\ b \rightarrow Skip$
$Action_C \mathrel{\widehat=} c \rightarrow Skip$
$\bullet\ Action_C;\ Action_A;\ Action_B$
end

The generated CSP_M script will have two files in this case. As there is no local DataType definition in this example, so `DataType.csp` file will not be generated.

The generated file, `ChannelDecls.csp` has the following channel declarations:

```
channel a
channel b
channel c
```

The final translated CSP_M script contained in `finalProcDecl.csp` is as follows:

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

9.3.3 Example 3 – Main Action having Internal Choice

Here, the final transformation of the process is based on the distributive property of sequential composition (;) and internal choice ($\sqcap$). This property can be stated as:
$A;\ (B \sqcap C) = (A;\ B) \sqcap (A;\ C)$

channel a, b, c
process $ProcEx2 \mathrel{\widehat=}$ **begin**
state

Ex2State

$x, y : \mathbb{N}$

$InitProcEx2 \mathrel{\widehat=} x := 1;\ y := 1$
$Action_A \mathrel{\widehat=} x := 0;\ a \rightarrow Skip$
$Action_B \mathrel{\widehat=} y := x + 1;\ b \rightarrow Skip$
$Action_C \mathrel{\widehat=} c \rightarrow Skip$
$\bullet\ Action_A;\ (Action_B \sqcap Action_C)$
end

The output generated from circus2cspm has two files. The channel declarations are in the file `ChannelDecls.csp`.

```
channel a
channel b
channel c
```

The final transformed CSP_M script contained in `finalProcDecl.csp` is as follows:

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

Note: Here, actions $Action_A$, $Action_B$ are sequentially composed. Similarly, actions $Action_A$, $Action_C$ are sequentially composed, based on the law, $A;\ (B \sqcap C) = (A;\ B) \sqcap (A;\ C)$.

9.3.4 Example 4 – Varied Order of Action Calls in Example 3

This example has same operators used in the main action as in the previous example. The difference here is the different order of action calls in the main action.

channel a, b, c
process $ProcEx3 \widehat{=}$ **begin**
state

$Ex3State$
$x, y : \mathbb{N}$

$InitProcEx3 \widehat{=} x := 1;\ y := 1$
$Action_A \widehat{=} x := 0;\ a \rightarrow Skip$
$Action_B \widehat{=} y := x + 1;\ b \rightarrow Skip$
$Action_C \widehat{=} c \rightarrow Skip$
$\bullet\ Action_B;\ (Action_C \sqcap Action_A)$
end

The generated file `ChannelDecls.csp` will have the channel declarations as follows:

```
channel a
channel b
channel c
```

The final transformed CSP_M script contained in `finalProcDecl.csp` is as follows:

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

Note: Here, the law governing distributivity of sequential composition and internal choice is used:
$B;\ (C \sqcap A) = (B;\ C) \sqcap (B;\ A)$
Note: Here, actions $Action_B$, $Action_C$ are sequentially composed. Similarly, actions $Action_B$, $Action_A$ are sequentially composed.

9.3.5 Example 5 – Multiple Assignments in an Action

This example contains a simple sequential composition case between the calling actions in the main action. The difference is the introduction of multiple assignment commands in the definition of $Action_C$.

channel a, b, c
process $ProcEx4 \widehat{=}$ **begin**
state

$Ex4State$
$x, y : \mathbb{N}$

$InitProcEx4 \widehat{=} x := 1;\ y := 1$
$Action_A \widehat{=} x := 0;\ a \rightarrow Skip$
$Action_B \widehat{=} y := x + 1;\ b \rightarrow Skip$
$Action_C \widehat{=} x := y - 1;\ y := y + x;\ x := 4;\ y := 5;\ c \rightarrow Skip$
$\bullet\ Action_A;\ Action_B;\ Action_C$
end

The generated file `ChannelDecls.csp` will have the channel declarations as follows:

```
channel a
channel b
channel c
```

The final transformed CSP_M script contained in `finalProcDecl.csp` is as follows:

```
ACTION_A(y,x)
= a->ACTION_B(y,0 )
ACTION_B(y,x)
= b->ACTION_C_0(x + 1 ,x)
ACTION_C_0(y,x) = ACTION_C_1(y,y - 1 )
ACTION_C_1(y,x) = ACTION_C_2(y + x,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
```

In case of multiple assignment commands in one action, the action name is indexed from 0 dealing with one assignment at a time and keeps increasing the index for each assignment.

9.3.6 Example 6 – Call of Actions having Sequential Composition and External Choice

In this example, the calling actions in main contains sequential composition and external choice.

channel a, b, c
process $ProcEx5 \widehat{=}$ **begin**
state

Ex5State

$x, y : \mathbb{N}$

$InitProcEx5 \mathrel{\widehat=} x := 1;\ y := 1$
$Action_A \mathrel{\widehat=} x := 0;\ a \rightarrow Skip$
$Action_B \mathrel{\widehat=} y := x + 1;\ b \rightarrow Skip$
$Action_C \mathrel{\widehat=} c \rightarrow Skip$
$\bullet\ Action_A;\ (Action_B \Box Action_C)$
end

The generated file `ChannelDecls.csp` will have the channel declarations as follows:

```
channel a
channel b
channel c
```

The final transformed CSP_M script contained in `finalProcDecl.csp` is as follows:

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

Note: Here, the case of main action with a combination of sequential composition and external choice is translated.

9.3.7 Example 7 – Call of Actions having External Choice and Sequential Composition

In this example, the main action contains action calls arranged like $(A \Box B);\ C$.

channel a, b, c
process $ProcEx6 \mathrel{\widehat=}$ **begin**
state

Ex6State

$x, y : \mathbb{N}$

$InitProcEx6 \mathrel{\widehat=} x := 1;\ y := 1$
$Action_A \mathrel{\widehat=} x := 0;\ a \rightarrow Skip$
$Action_B \mathrel{\widehat=} y := x + 1;\ b \rightarrow Skip$
$Action_C \mathrel{\widehat=} c \rightarrow Skip$
$\bullet\ (Action_A \Box Action_B);\ Action_C$
end

The generated file `ChannelDecls.csp` will have the channel declarations as follows:

```
channel a
channel b
channel c
```

The final transformed CSP_M script contained in `finalProcDecl.csp` is as follows:

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

Note: In this case the distributive property of external choice and sequential composition holds, i.e. $(A \Box B);\ C = (A;\ C) \Box (B;\ C)$. Consequently, in the output, $Action_A$ and $Action_C$ are sequentially composed on the left side of external choice. Similarly, $Action_B$ and $Action_C$ are sequentially composed on the right side of external choice.

9.3.8 Example 8 – Including Output Prefixing Action

In this example, an output channel is there in the definition of $Action_C$. The expression $x+y$ is calculated and given as output on channel c.

channel a, b
channel $c : \mathbb{N}$
process $ProcEx7 \mathrel{\widehat=}$ **begin**
state

Ex7State
$x, y : \mathbb{N}$

$Action_A \mathrel{\widehat=} x := 0;\ a \rightarrow Skip$
$Action_B \mathrel{\widehat=} y := x + 1;\ b \rightarrow Skip$
$Action_C \mathrel{\widehat=} c!(x+y) \rightarrow Skip$
$\bullet\ Action_A;\ (Action_B \Box Action_C)$
end

The generated file `ChannelDecls.csp` will have the channel declarations as follows:

```
channel a
channel b
channel c : {0,1,2,3,4,5}
```

The final transformed CSP_M script contained in `finalProcDecl.csp` is as follows:

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

Note: Here, channel c outputs a value of type natural number. So, in the declaration of this particular channel i.e. channel c in this case, a list of possible values has to be given. Here we assume that the possible range of numbers is between 0 and +5.

9.4 Summary

This chapter first evaluated the translated examples using the SimpleCircus2CSPM tool and gave comments on the output acquired for each translated example. It also described the case study of the cache coherence protocol [FS96] specified in the notation of Simple*Circus*. Subsequently, the *Circus* examples translated using the tool circus2cspm are discussed. The next chapter concludes the thesis by summarising the technical contribution during each phase of our research.

Chapter 10

Conclusions

Here, we conclude this thesis by presenting the technical contributions achieved:

10.1 Extending a *CSP* Model of Flash Device Behaviour

As an intial case study and to gain hands-on experience of CSP_M and the model checker FDR, the earlier work on a *CSP* model for flash device behaviour [Cat08, BOC09] was extended.

In [BOC09], the Open Nand Flash Model (ONFi 1.0) was first represented in State Chart XML(SC-XML). The processes in the model had a direct correspondence between the state machine of the particular process and its entry in the state-chart XML. Then, using XSLT (a translator for XML), SC-XML descriptions of state machines were translated into their equivalent HTML pages and CSP_M sources. This automatic translation helped in modelling flash device behaviour because the call to each process in CSP_M required establishing a long list of parameters. In [BOC09], tests on the flash model were performed using FDR trace refinement checking, which is a weak testing mode. Also, this testing was performed only on the mandatory portion of the ONFi model. The testing could not be performed on the full ONFi 1.0 model.

So, the following enhancements to the original model were achieved:

1. Upgrading of the ONFi 1.0 model to an ONFi 2.1 model (which was the latest at that time) was done. The SC-XML description of the model was updated. Using XSLT, as before, HTML pages and CSP_M sources were generated.

2. The tests were performed on the model with more rigorous checks using Failures Refinement and Failures-Divergence Refinement. In the old version, the testing was done with the weaker notion of trace refinement only.

3. Through the technique of pushing all use of the hiding operators down to the deepest level possible, as well as the compression techniques available in the FDR toolkit, the state-space of the model was much reduced.

The work for the *CSP* model of flash device behaviour can be extended in the following direction:
Although having achieved more rigorous testing and state-space compression of the model, the full ONFi 2.1 model could not be compiled. These are the possibilities:

- One possible solution for the full model compilation is to try it on a larger machine with more physical and virtual memory, to handle the high demands made by initial ISM generation.

- Another way for dealing with this problem is to analyse the model to see if it can be re-factored into independent chunks.

- A third possibility is trying to use the relatively recently released FDR3 to see if it is better able to cope.

This work has been published in conference proceedings [BB10b]. The body of the paper is included in Chapter 4, on page 25.

10.2 Development of a Prototype to Translate from *Circus* to CSP_M using Java

In order to target the research objective of translating from *Circus* to CSP_M [BB10a], the research went through a number of steps.

The first attempt was made to modify the sources of JCircus [dF05] to get the translation working from *Circus* to CSP_M. Here, the tool development was in the imperative language, Java. The original sources were developed to translate from *Circus* to Java. As *Circus* specifications are parsed and type-checked through the JCircus tool, it was the natural choice to re-use these sources. Here, we modified the sources of JCircus to produce the translation from *Circus* to CSP_M. The prototype tool developed is named circus2cspm. The examples translated using the tool circus2cspm are included in Section 9.3, on page 93. These examples contain the sequential composition, assignment commands, prefixing actions, if statements, basic constructs and some binary and equality check operators in a guard. A number of combinations of calling actions in the main action were tested to verify the final transformation of the process definition. The `TranslatorVisitor` and `Translator2CSP` java classes were significantly modified. All the outputs of CSP_M generated from circus2cspm were tested with the ProBE tool and FDR for correct parsing and type-checking.

Despite having the initial implementation of the translator, as described above in Java being quite straightforward, the other operators of the language were found to be hard to implement in that structure and in the implementation language i.e. Java. We found the development in Java quite complex, especially in the record keeping process and moving from initial to intermediate and then from intermediate to the final transformation. In the examples considered, we dealt with some simple action compositions. But as the constructs became more complex in form, it was increasingly hard to implement it. Pattern matching compositions of complex combinations of actions proved very difficult to implement in the present state of the source code of circus2cspm. This pattern matching was required to deal with the implementation of all the *CSP* laws in the main action. This goal was found to be hard to achieve, using the implementation in Java.

The work of Java implementation might have the following future directions:

- The tool circus2cspm is a prototype for the translation. One way of extension is using the updated sources of the tool JCircus 2.0 presented in the work [BO12].
- As a result of later translation work for this thesis, the translation steps for the target languages are presented in a better formalised way. The Haskell implementation also improved the way the translation is done. Now, re-structuring circus2cspm Java code might improve its capability.
- After re-structuring of the circus2cspm code suggested above, the implementation of more language constructs is a future work.

10.3 Mathematical Proofs of Semantic Justification for the proposed Translation using the Unifying Theories of Programming (UTP) Semantics

Now, we discuss the achievement acquired in the mathematical foundation of the translation strategy between the target languages. First of all, the semantics of *Circus* and CSP_M in the Unifying Theories of Programming

(UTP) framework are specified for the selected constructs of the languages. The chosen subset of the original *Circus* and *CSP* languages are named as Simple*Circus* and Simple*CSP*. The semantical difference between the two is captured by the need for extra observation variables i.e. $state, state'$, in case of Simple*Circus*. A formal link was proposed to connect the two theories. We proposed theorem 6.2.1, on page 47 for the linking pairs between the two languages. If P_X is a Simple*Circus* program and P_C is a Simple*CSP* program, then the transformation predicate ($\mathscr{T}$) will translate between the two.

For all Simple*Circus* programs P_X, we define a link to the *CSP* world that hides the state. In effect, we view *Circus* and *CSP* processes as equivalent, if the translation is a refinement:

$$\mathscr{T}(P_X) \sqsubseteq (\exists state, state' \bullet P_X)$$

By mathematical proofs included in Chapter 6, it is proved that this linking predicate preserves the semantics of most of the language operators, with the notable exception of sequential composition. We applied the theorem 6.2.1 for the constructs considered in Simple*Circus* and Simple*CSP*. The established link demonstrated the feasibility of the translation between the two languages, and justifies the close attention paid to translating sequential composition. This is novel work.

The work of mathematical proofs of the link might have the following future directions:

- The future enhancement in this area is to consider the remaining constructs of the target languages.
- The proofs of link are done manually. Saoithin [But10] is a theorem prover designed to support the UTP framework and do the proofs in an equational style. The mechanisation of the link proofs in Saoithin is a possible future direction of work.

The translation process from *Circus* to CSP_M is divided into steps. In fact, each translation step is a function, typically defined by pattern matching against the range of syntax forms. These steps are formalised and mathematically represented in Chapter 7, on page 57. Once formally specified, the mathematical representation made the implementation phase much easier and straightforward.

10.4 Using Haskell for Development of the Prototype for *Circus* to CSP_M Translation

The prototype tool called SimpleCircus2CSPM is implemented in the functional language, Haskell. This is a novel approach of prototyping the translation from *Circus* to CSP_M, exploiting the expressiveness of functional languages. In the Haskell implementation, a number of examples are provided. We covered a number of constructs in our prototype such as prefix actions, sequential composition, internal choice, external choice, hiding action, interface parallel, alphabetised parallel, conditional, assignment command, calling an action, as well as their generalised forms. A significant number (forty four) of *CSP* algebraic laws are formalised in the implementation. A normalisation process is also implemented in the prototype. A simple implementation of pretty printing is also provided which can be extended to provide a more sophisticated one. The formalised translation steps defined in Chapter 7 are implemented in Haskell code. The examples translated using the tool are listed in Section 9.1.

The notation of Simple*Circus* developed for the SimpleCircus2CSPM tool is used to specify a cache coherence protocol given in [FS96]. This is included in Section 9.2. All the processes for processor and memory specified in [FS96] are re-stated using Simple*Circus* notation. One can pretty print each process of processor and memory specified through the prototype. This provides evidence for the utility of the Simple*Circus* notation for performing industry case studies in the future.

10.5 Summary

Now, we summarise the overall contribution of the thesis.

To make use of the available and industry-proven tools for a particular programming paradigm, there is a need to develop a formally verified link between one world and the other. The aim of this work was to develop a formally verified translation between the state-rich process algebra i.e. *Circus* to the state-poor process algebra i.e. *CSP*. For developing a link between targeted formal languages, the key translations required between the two languages are identified. For ensuring correctness of the translation, the key translation / refinement steps are formalised. This formed the theoretical core of the work and supported the soundness of the link. The prototyping of the translation strategy by exploiting the expressiveness of the functional language, Haskell is a novel approach. Providing the mathematical foundation by giving semantical justification for the translation between the target languages is another major contribution of the thesis.

Bibliography

[ARA04] M. Al-Rousan and S. Ahmed. Implementation of cache coherence protocol for COMA multiprocessor systems based on the scalable coherent interface. *Computer Standards & Interfaces*, 27(1):71–88, 2004.

[BB10a] Arshad Beg and Andrew Butterfield. Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation. In *Proceedings of International Conference on Frontiers of Information Technology, Islamabad, Pakistan*, number 47, pages 1–5. ACM, 2010.

[BB10b] Arshad Beg and Andrew Butterfield. Modelling flash devices with FDR: Progress and limits. In *Proceedings of International Conference on Frontiers of Information Technology, Islamabad, Pakistan*, number 18, pages 1–6. ACM, 2010.

[BFW09] Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program*, 74(4):219–237, 2009.

[BGW09] Andrew Butterfield, Pawel Gancarski, and Jim Woodcock. State visibility and communication in unifying theories of programming. In Wei-Ngan Chin and Shengchao Qin, editors, *Third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 47–54. IEEE Computer Society, 2009.

[BH99] Jonathan P. Bowen and Jifeng He. Hardware Compilation: Verification and Rapid-prototyping. RUCS Technical Report RUCS/1999/TR/012/A, Department of Computer Science, University of Reading, October 1999.

[BH01] Jonathan P. Bowen and Jifeng He. An approach to the specification and verification of a hardware compilation scheme. *The Journal of Supercomputing*, 19(1):23–39, 2001.

[BM10] Mikkel Bundgaard and Robin Milner. Unfolding CSP. Springer, 2010.

[BO12] S. L. M. Barrocas and M. V. M. Oliveira. JCircus 2.0: an Extension of an Automatic Translator from Circus to Java. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Baekgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 15–36, August 2012.

[BOC09] Andrew Butterfield and Art Ó Catháin. Concurrent models of flash memory device behaviour. *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19–21, 2009, Revised Selected Papers*, pages 70–83, 2009.

[BOG02] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, July 2002.

[Bro08] Neil C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, S. Stepney, F. A. C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, pages 67–83. IOS Press, September 2008.

[Bro09] Neil C. C. Brown. Automatically generating CSP models for Communicating Haskell Processes. *Journal of Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, Vol. 23, 2009.

[BSW07] Andrew Butterfield, Adnan Sherif, and Jim Woodcock. Slotted-Circus. In Jim Davies and Jeremy Gibbons, editors, *6th International Conference on Integrated Formal Methods (IFM)*, volume 4591 of *Lecture Notes in Computer Science*, pages 75–97. Springer, 2007.

[But10] Andrew Butterfield. Saoithin: A theorem prover for UTP. In Shenchao Qin, editor, *Unifying Theories of Programming, Third International Symposium, UTP 2010, Shanghai, China*, volume 6445 of *Lecture Notes in Computer Science*, pages 137–156, Shanghai, China, November 2010. Springer.

[BW07] Andrew Butterfield and Jim Woodcock. Formalising flash memory: First steps. In *12th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 251–260. IEEE Computer Society, 2007.

[Cat08] Art Ó Catháin. Modelling flash memory device behaviour using CSP. Taught M.Sc dissertation, School of Computer Science and Statistics, Trinity College Dublin, 2008. Also published as Technical Report TCD-CS-2008-47.

[Cav97] A. L. C. Cavalcanti. *A Refinement Calculus for Z.* PhD thesis, Oxford University Computing Laboratory, Oxford, 1997.

[CH13] A. L. C. Cavalcanti and Robert M. Hierons. Testing with inputs and outputs in CSP. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 359–374. Springer-Verlag, 2013.

[CHW06] A. L. C. Cavalcanti, Will Harwood, and Jim Woodcock. Pointers and records in the Unifying Theories of Programming. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming (UTP)*, volume 4010 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2006.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CMP04] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In Alan J. Hu and Andrew K. Martin, editors, *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2004.

[CMW90] Russell M. Clapp, Trevor N. Mudge, and Donald C. Winsor. Cache coherence requirements for interprocess rendezvous. *International Journal of Parallel Programming*, 19(1):31–51, February 1990.

[CW04] A. L. C. Cavalcanti and Jim Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock, editors, *First Pernambuco Summer School on Software Engineering (PSSE)*, volume 3167 of *Lecture Notes in Computer Science*, pages 220–268. Springer, 2004.

[dF05] Angela Figueiredo de Freitas. From Circus to Java: Implementation and verification of a translation strategy. Master's thesis, University of York, December, 2005.

[DNA05] Nirav Dave, Man Cheuk Ng, and Arvind. Automatic synthesis of cache-coherence protocol processors using bluespec. In *Third ACM-IEEE International Conference on. Formal Methods and Models for Codesign, (MEMOCODE)*, pages 25–34. IEEE, 2005.

[EK03] E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2003.

[EL02] Lars-Henrik Eriksson and Peter A. Lindsay, editors. *Mechanical Abstraction of CSP_Z Processes*, volume 2391 of *Lecture Notes in Computer Science*. Springer, 2002.

[FO09] Miguel Alexandre Ferreira and José Nuno Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19–21, 2009 Revised Selected Papers*, pages 153–169, 2009.

[For05] Formal Systems (Europe) Ltd. *FDR Manual*, June 2005.

[FS96] Amy Felty and Frank Stomp. A correctness proof of a cache coherence protocol. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 128, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[FSO08] M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying intel flash file system core specification. In P.G. Larsen J.S. Fitzgerald and S. Sahara, editors, *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, pages 54–71, School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.

[FW09] Leo Freitas and Jim Woodcock. FDR explorer. *Formal Asp. Comput*, 21(1-2):133–154, 2009.

[GB09a] Paweł Gancarski and Andrew Butterfield. The denotational semantics of *slotted-circus*. In Ana Cavalcanti and Dennis Dams, editors, *FM2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 451–466. Springer, 2009.

[GB09b] Pawel Gancarski and Andrew Butterfield. Slotted circus a generic UTP framework for discretely-timed Circus. Technical report, Department of Computer Science, Trinity College Dublin, 2009.

[GS97] Andy Galloway and Bill Stoddart. An operational semantics for ZCCS. In *First IEEE International Conference on Formal Engineering Methods (ICFEM)*, page 272. IEEE Computer Society, 1997.

[H^+06] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 1.0, ONFI, www.onfi.org, 28th December 2006.

[H^+09] Hynix Semiconductor et al. Open NAND Flash Interface Specification. Technical Report Revision 2.1, ONFI, www.onfi.org, 14th January 2009.

[HH98] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall International, Englewood Cliffs, NJ, 1998.

[HH05] Jifeng He and C. A. R. Hoare. Linking theories of concurrency. In Dang Van Hung and Martin Wirsing, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 3722 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2005.

[HM05] C. A. R. Hoare and Jayadev Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005.

[HMLS09] C. A. R. Hoare, Jayadev Misra, Gary T. Leavens, and Natarajan Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv*, 41(4), 2009.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 2004.

[ISK06] Syed M. S. Islam, Mohammed H. Sqalli, and Sohel Khan. Modeling and formal verification of DHCP using SPIN. *International Journal of Computer Science and Applications (IJCSA)*, 3(2):145–159, 2006.

[JH05] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. In *Proc. Verified Software: Theories, Tools, Experiments (VSTTE), Zürich*, 2005.

[KCKK08a] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver – an experience report. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *SPIN*, volume 5156 of *Lecture Notes in Computer Science*, pages 144–159. Springer, 2008.

[KCKK08b] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Pre-testing flash device driver through model checking techniques. In *IEEE International Conference on Software Testing Verification and Validation (ICST)*, pages 475–484. IEEE Computer Society, 2008.

[KJ08] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *Conference on Abstract State Machines, Alloy, B and Z (ABZ)*, volume 5238 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2008.

[KJ09] Eunsuk Kang and Daniel Jackson. Designing and analyzing a flash file system with Alloy. *International Journal of Software and Informatics (IJSI) 2009*, Vol 3, No. 1, 2009.

[KK09] Moonzoo Kim and Yunho Kim. Concolic testing of the multi-sector read operation for flash memory file system. *Formal Methods: Foundations and Applications: 12th Brazilian Symposium on Formal Methods, SBMF 2009 Gramado, Brazil, August 19–21, 2009 Revised Selected Papers*, pages 251–265, 2009.

[KKK08] Moonzoo Kim, Yunho Kim, and Hotae Kim. Unit testing of flash memory device driver through a SAT-based model checker. In *IEEE Proceedings of Automated Software Engineering (ASE)*, pages 198–207. IEEE, 2008.

[MAF08] J. N. Oliveira M. A. Ferreira, S. S. Silva. Verifying Intel Flash File System Core Specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, 2008.

[MFMU05] Tim Miller, Leo Freitas, Petra Malik, and Mark Utting. CZT support for Z extensions. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Fifth International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *Lecture Notes in Computer Science*, pages 227–245. Springer, 2005.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[MS01] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Sci. Comput. Program*, 40(1):59–96, 2001.

[NG02] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. *Formal Methods in System Design*, 20(1):107–125, 2002.

[OC04] M. V. M. Oliveira and A. L. C. Cavalcanti. From Circus to JCSP. In *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 320 – 340. Springer-Verlag, November 2004.

[OCW06] M. V. M. Oliveira, A. L. C. Cavalcanti, and Jim Woodcock. Unifying theories in ProofPower-Z. In Steve Dunne and Bill Stoddart, editors, *UTP*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.

[OCW09] M. V. M. Oliveira, A. L. C. Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Asp. Comput*, 21(1-2):3–32, 2009.

[Oli05] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, University of York, 2005.

[PD97] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *CSURV: Computing Surveys*, 29, 1997.

[PS04] Jonathan D. Phillips and G. S. Stiles. An automatic translation of CSP to Handel-C. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*. IOS Press, Amsterdam, 2004.

[QGPY08] WanXia Qu, Yang Guo, Zhengbin Pang, and Xiaodong Yang. Efficient verification of parameterized cache coherence protocols. In *9th International Conference for Young Computer Scientists (ICYCS)*, pages 154–159. IEEE Computer Society, 2008.

[Rep] `https://bitbucket.org/andrewbutterfield/csp-in-haskell/overview`, accessed at 13:08pm, 17th July, 2014.

[RMK03] Abhik Roychoudhury, Tulika Mitra, and S. R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 10828–10833. IEEE Computer Society, 2003.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, New York, 1998. Oxford.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[SCS06] Thiago L. V. L. Santos, Ana Cavalcanti, and Augusto Sampaio. Object-orientation in the UTP. In Steve Dunne and Bill Stoddart, editors, *UTP*, volume 4010 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2006.

[SD95] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. *Lecture Notes in Computer Science*, 987:21–34, 1995.

[She00] Xiaowei Shen. *Design and verification of adaptive cache coherence protocols*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2000.

[SPH+00] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, Anne E. Condon, Milo M. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. Technical Report 1412, Univ. of Wisconsin Computer Sciences, Madison, WI, March 2000.

[ST05] Steve Schneider and Helen Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.

[TG08] Lizeth Tapia and Chris George. Model checking concurrent RSL with CSP_M and FDR2. Technical Report UNU-IIST Report No. 393, International Institute for Software Technology, The United Nations University, May 2008.

[URLa] `http://www.sundoginteractive.com/sunblog/posts/top-ten-most-infamous-software-bugs-of-all-time/`, accessed at 16:40pm, 18th September, 2013.

[URLb] `http://www.zdnet.com/the-top-10-it-disasters-of-all-time-3039290976`, accessed at 12:40pm, 2nd October, 2013.

[URLc] `http://en.wikipedia.org/wiki/formal_methods`, accessed at 13:15pm, 31st of august, 2012.

[URLd] `http://en.wikipedia.org/wiki/internet`, accessed at 14:13pm, 20th January, 2015.

[URLe] `http://en.wikipedia.org/wiki/mars_climate_orbiter`, accessed at 13:16pm, 2nd October, 2013.

[URLf] `http://www.cs.york.ac.uk/circus/`, accessed at 15:33pm, 12th March, 2013.

[URLg] `http://www.nasa.gov`, accessed at 14:33pm, 20th January, 2015.

[VG08] Abigail Parisaca Vargas and Chris George. Formalisitng the translation from RSL to CSP. Technical Report UNU-IIST Report No. 395, International Institute for Software Technology, The United Nations University, May 2008.

[WC02] Jim Woodcock and A. L. C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.

[WCF05] Jim Woodcock, A. L. C. Cavalcanti, and Leonardo Freitas. Operational semantics for model checking Circus. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2005.

[WD96] Jim Woodcock and Jim Davies. *Using Z Specification, Refinement and Proof*. Prentice Hall International Series in Computer Science, 1996.

[WE04] Kylie Williams and Robert Esser. Verification of the futurebus+ cache coherence protocol: A case study in model checking. In Vladimir Estivill-Castro, editor, *Twenty-Seventh Australasian Computer Science Conference (ACSC2004)*, volume 26 of *Conferences in Research and Practice in Information Technology (CRPIT)*, pages 65–71, Dunedin, New Zealand, 2004. ACS.

[Wil09] Matthew Wilson. Quality matters: Correctness, robustness and reliability. *Overload*, 17(93), October 2009. What do we mean by quality? Matthew Wilson considers some definitions.

[Woo06] Jim Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.

[YP07] Letu Yang and Michael Poppleton. Automatic translation from combined B and CSP specification to Java programs. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2007.

Appendix A

Introducing Karnaugh Maps for Graphical Proofs

A.1 Graphical Approach

This tutorial for proof through graphical approach is for reference and is used in the proof of theorem B.1.1. The approach is proposed and devised by Dr. Andrew Butterfield. The need for developing this approach arose when the theorem B.1.1 proof proved to be too complicated using the conventional technique of expanding the definitions.

A.1.1 Standard Reactive Diagram

In these diagrams we use 1 and . to stand for *True* and *False* respectively, for compactness. These diagrams are only valid for use as described below when the predicates inside the boxes do not have $ok, ok', wait, wait'$ among their free variables.

Given the following shorthand,

$$A_{bcde} \quad \widehat{=} \quad A[b,c,d,e/ok,ok',wait,wait']$$

then the following is true:

$$P \quad \equiv \quad \left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & P_{0000} & P_{0001} & P_{0011} & P_{0010} & \neg o' \\ \hline \neg o & P_{0100} & P_{0101} & P_{0111} & P_{0110} & o' \\ \hline o & P_{1100} & P_{1101} & P_{1111} & P_{1110} & o' \\ \hline o & P_{1000} & P_{1001} & P_{1011} & P_{1010} & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right)$$

A.1.2 Logical Operations

We illustrate by example.

Logical-And

A square with a dot (.) in either argument (*False*) is a dot in the result, otherwise we connect with $\wedge$

	$\neg w$	$\neg w$	w	w	
$\neg o$	A	B	C	D	$\neg o'$
$\neg o$	E	F	G	H	o'
o	.	.	.	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$\wedge$

	$\neg w$	$\neg w$	w	w	
$\neg o$	.	P	.	T	$\neg o'$
$\neg o$	.	Q	.	U	o'
o	.	R	.	V	o'
o	.	S	.	W	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$\equiv$

	$\neg w$	$\neg w$	w	w	
$\neg o$	.	$B \wedge P$	.	$D \wedge T$	$\neg o'$
$\neg o$	.	$F \wedge Q$	.	$H \wedge U$	o'
o	.	.	.	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

Logical-Or

A square with a dot (.) in both argument (*False*) is a dot in the result, otherwise we connect with $\vee$

	$\neg w$	$\neg w$	w	w	
$\neg o$	A	B	C	D	$\neg o'$
$\neg o$	E	F	G	H	o'
o	.	.	.	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$\vee$

	$\neg w$	$\neg w$	w	w	
$\neg o$	.	P	.	T	$\neg o'$
$\neg o$	.	Q	.	U	o'
o	.	R	.	V	o'
o	.	S	.	W	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$\equiv$

	$\neg w$	$\neg w$	w	w	
$\neg o$	A	$B \vee P$	C	$D \vee T$	$\neg o'$
$\neg o$	E	$F \vee Q$	G	$H \vee U$	o'
o	.	R	.	V	o'
o	.	S	.	W	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

Logical-Not

Just negate it all, remembering that . (*False*) becomes 1 (*True*).

$\neg$

	$\neg w$	$\neg w$	w	w	
$\neg o$	A	B	C	D	$\neg o'$
$\neg o$	E	F	G	H	o'
o	.	.	.	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$\equiv$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$\neg A$	$\neg B$	$\neg C$	$\neg D$	$\neg o'$
$\neg o$	$\neg E$	$\neg F$	$\neg G$	$\neg H$	o'
o	1	1	1	1	o'
o	1	1	1	1	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

Implication

A square with a dot (.) in the first becomes 1 (*True*). Squares with a value only from the first get negated, while those values in the second where there is a 1 in the first only are left as is. Squares with values from both are connected with $\Rightarrow$.

$$\left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & A & B & C & D & \neg o' \\ \hline \neg o & E & F & G & H & o' \\ \hline o & . & . & . & . & o' \\ \hline o & . & . & . & . & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right) \Rightarrow \left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & . & P & . & T & \neg o' \\ \hline \neg o & . & Q & . & U & o' \\ \hline o & . & R & . & V & o' \\ \hline o & . & S & . & W & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right)$$

$$\equiv$$

$$\left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & \neg A & B \Rightarrow P & \neg C & D \Rightarrow T & \neg o' \\ \hline \neg o & \neg E & F \Rightarrow Q & \neg G & H \Rightarrow U & o' \\ \hline o & 1 & 1 & 1 & 1 & o' \\ \hline o & 1 & 1 & 1 & 1 & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right)$$

Equivalence

A square with a dot (.) in both becomes 1 (*True*). Squares with a value only from one or the other get negated. Squares with values from both are connected with $\equiv$.

$$\left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & A & B & C & D & \neg o' \\ \hline \neg o & E & F & G & H & o' \\ \hline o & . & . & . & . & o' \\ \hline o & . & . & . & . & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right) \equiv \left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & . & P & . & T & \neg o' \\ \hline \neg o & . & Q & . & U & o' \\ \hline o & . & R & . & V & o' \\ \hline o & . & S & . & W & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right)$$

$$\equiv$$

$$\left(\begin{array}{c|c|c|c|c|c} & \neg w & \neg w & w & w & \\ \hline \neg o & \neg A & B \equiv P & \neg C & D \equiv T & \neg o' \\ \hline \neg o & \neg E & F \equiv Q & \neg G & H \equiv U & o' \\ \hline o & 1 & \neg R & 1 & \neg V & o' \\ \hline o & 1 & \neg S & 1 & \neg W & \neg o' \\ \hline & \neg w' & w' & w' & \neg w' & \end{array}\right)$$

A.1.3 Conditionals as Projections

Conditional statements with $ok, ok', wait, wait'$ in the conditions act as projection operations over these graphs.

$$\left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & A & B & C & D & \neg o' \\ \hline
\neg o & E & F & G & H & o' \\ \hline
o & I & J & K & L & o' \\ \hline
o & M & N & O & P & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right)$$

$$=$$

$$\left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & . & . & . & . & \neg o' \\ \hline
\neg o & . & . & . & . & o' \\ \hline
o & I & J & K & L & o' \\ \hline
o & M & N & O & P & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right) \lhd ok \rhd \left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & A & B & C & D & \neg o' \\ \hline
\neg o & E & F & G & H & o' \\ \hline
o & . & . & . & . & o' \\ \hline
o & . & . & . & . & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right)$$

$$=$$

$$\left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & . & . & . & . & \neg o' \\ \hline
\neg o & E & F & G & H & o' \\ \hline
o & I & J & K & L & o' \\ \hline
o & . & . & . & . & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right) \lhd ok' \rhd \left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & A & B & C & D & \neg o' \\ \hline
\neg o & . & . & . & . & o' \\ \hline
o & . & . & . & . & o' \\ \hline
o & M & N & O & P & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right)$$

$$=$$

$$\left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & . & . & C & D & \neg o' \\ \hline
\neg o & . & . & G & H & o' \\ \hline
o & . & . & K & L & o' \\ \hline
o & . & . & O & P & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right) \lhd wait \rhd \left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & A & B & . & . & \neg o' \\ \hline
\neg o & E & F & . & . & o' \\ \hline
o & I & J & . & . & o' \\ \hline
o & M & N & . & . & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right)$$

$$=$$

$$\left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & . & B & C & . & \neg o' \\ \hline
\neg o & . & F & G & . & o' \\ \hline
o & . & J & K & . & o' \\ \hline
o & . & N & O & . & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right) \lhd wait' \rhd \left(\begin{array}{c|c|c|c|c|c}
 & \neg w & \neg w & w & w & \\ \hline
\neg o & A & . & . & D & \neg o' \\ \hline
\neg o & E & . & . & H & o' \\ \hline
o & I & . & . & L & o' \\ \hline
o & M & . & . & P & \neg o' \\ \hline
 & \neg w' & w' & w' & \neg w' &
\end{array}\right)$$

A.1.4 Examples

DIV $\neg ok \wedge tr \leq tr'$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$\neg o'$
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	o'
o	.	.	.	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

OKID $ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref$

	$\neg w$	$\neg w$	w	w	
$\neg o$	.	.	.	.	$\neg o'$
$\neg o$	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

II $DIV \vee OKID$

$$tr \leq tr' \quad \equiv \quad tr \leq tr' \vee tr' = tr \wedge ref' = ref$$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$\neg o'$
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	o'
o	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

B $wait' \wedge tr' = tr \vee tr < tr'$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	$\neg o'$
$\neg o$	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

B' $tr' = tr \lhd wait' \rhd tr < tr'$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	$\neg o'$
$\neg o$	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

J $(ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	$\neg o'$
$\neg o$	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

P ⊢ Q $ok \wedge P \Rightarrow ok' \wedge Q$

$$\begin{aligned} ok, ok' &\notin P, Q \\ A_{cd} &\mathrel{\widehat{=}} A[c, d/wait, wait'] \end{aligned}$$

	$\neg w$	$\neg w$	w	w	
$\neg o$	1	1	1	1	$\neg o'$
$\neg o$	1	1	1	1	o'
o	$P_{00} \Rightarrow Q_{00}$	$P_{01} \Rightarrow Q_{01}$	$P_{11} \Rightarrow Q_{11}$	$P_{10} \Rightarrow Q_{10}$	o'
o	$\neg P_{00}$	$\neg P_{01}$	$\neg P_{11}$	$\neg P_{10}$	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

A.1.5 Lemma 1 – Graphical Approach

Lemma A.1.1

$$II_C \wedge B = ?$$

Proof Strategy: Graphical Proof.

B

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	$\neg o'$
$\neg o$	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

II

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$\neg o'$
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	o'
o	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$II \wedge B$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	$\neg o'$
$\neg o$	$tr < tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr < tr'$	o'
o	.	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

From this we can read-off:

$$\begin{array}{ll} & \neg ok \wedge \neg wait' \wedge tr < tr' \\ \vee & \neg ok \wedge wait' \wedge tr \leq tr' \\ \vee & ok \wedge ok' \wedge wait \wedge wait' \wedge tr' = tr \wedge ref' = ref \end{array}$$

■

A.1.6 Lemma 2 – Graphical Approach

Lemma A.1.2

$$II_C \wedge B' = ?$$

Proof Strategy: Graphical Proof.

B'

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	$\neg o'$
$\neg o$	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	o'
o	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

II

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$\neg o'$
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	o'
o	$tr' = tr \wedge ref' = ref$	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

$II \wedge B'$

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	$\neg o'$
$\neg o$	$tr < tr'$	$tr = tr'$	$tr = tr'$	$tr < tr'$	o'
o	.	.	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

From this we can read-off:

$$\begin{array}{ll} & \neg ok \wedge \neg wait' \wedge tr < tr' \\ \vee & \neg ok \wedge wait' \wedge tr = tr' \\ \vee & ok \wedge ok' \wedge wait \wedge wait' \wedge tr' = tr \wedge ref' = ref \end{array}$$

∎

Appendix B

Proofs for the Link between Simple*Circus* and Simple***CSP***

B.1 A Graphical Proof Attempt

(with Andrew Butterfield)

Theorem B.1.1

$$(a \rightarrow_C Skip_C) = (\exists state, state' \bullet a \rightarrow_X Skip_X)$$

Proof Strategy: reduce LHS and RHS to same form.

LHS:

$$
\begin{array}{ll}
 & a \rightarrow_C Skip_C \\
= & \quad \text{“ def of } a \rightarrow Skip_C \text{ ”} \\
 & \mathbf{CSP1}(ok' \wedge do_A(a)) \\
= & \quad \text{“ def of } \mathbf{CSP1} \text{ ”} \\
 & ok' \wedge do_A(a) \vee DIV \\
= & \quad \text{“ def of } do_A(a) \text{ ”} \\
 & ok' \wedge \Phi(a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \vee DIV \\
= & \quad \text{“ def of } \Phi(A) \text{ ”} \\
 & ok' \wedge \mathbf{R}(a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \wedge B \vee DIV \\
= & \quad \text{“ re-arrange ”} \\
 & DIV \vee ok' \wedge B \wedge \mathbf{R}(a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \\
= & \quad \text{“ alternate expansion of } \mathbf{R} \text{ ”} \\
 & DIV \vee ok' \wedge B \wedge \big(I\!I_C \lhd wait \rhd \big((a \notin ref' \lhd wait' \rhd tr' = tr \frown \langle a \rangle) \wedge tr \leq tr'\big)\big) \\
= & \quad \text{“ Distribute in } tr \leq tr' \text{, subsume by } tr' = tr \frown _ \text{ ”} \\
 & DIV \vee ok' \wedge B \wedge \big(I\!I_C \lhd wait \rhd \big(a \notin ref' \wedge tr \leq tr' \lhd wait' \rhd tr' = tr \frown \langle a \rangle\big)\big) \\
= & \quad \text{“ Push } B \text{ into nested conditionals } (\wedge\text{-}\lhd\rhd\text{-distr}) \text{ ”} \\
 & DIV \vee ok' \wedge \big(B \wedge I\!I_C \lhd wait \rhd \big(B \wedge a \notin ref' \wedge tr \leq tr' \lhd wait' \rhd B \wedge tr' = tr \frown \langle a \rangle\big)\big)
\end{array}
$$

We pause here, to observe that we have interactions with B and conditionals involving $wait'$.

$$
\begin{aligned}
B \wedge wait' &= wait' \wedge tr \leq tr' \\
B \wedge \neg wait' &= \neg wait' \wedge tr < tr' \\
B' \wedge wait' &= wait' \wedge tr = tr' \\
B' \wedge \neg wait' &= \neg wait' \wedge tr < tr'
\end{aligned}
$$

So we can simplify B in the innermost conditional:

$$
\begin{aligned}
& DIV \vee ok' \wedge \left(B \wedge II_C \lhd wait \rhd \left(tr \leq tr' \wedge a \notin ref' \wedge tr \leq tr' \lhd wait' \rhd tr < tr' \wedge tr' = tr ^\frown \langle a \rangle\right)\right) \\
= \quad & \text{“ simplify, subsume ”} \\
& DIV \vee ok' \wedge \left(B \wedge II_C \lhd wait \rhd \left(a \notin ref' \wedge tr \leq tr' \lhd wait' \rhd tr' = tr ^\frown \langle a \rangle\right)\right)
\end{aligned}
$$

At this point we can see clearly that B is simply wrong, because when $wait$ is false (we are running) and $wait'$ is true (we are waiting to perform an event) we see that we assert only **R1** regarding tr and tr', whereas we should assert $tr' = tr$. So we should use B' instead:

$$
\begin{aligned}
& DIV \vee ok' \wedge \left(B' \wedge II_C \lhd wait \rhd \left(tr = tr' \wedge a \notin ref' \wedge tr \leq tr' \lhd wait' \rhd tr < tr' \wedge tr' = tr ^\frown \langle a \rangle\right)\right) \\
= \quad & \text{“ subsume twice ”} \\
& DIV \vee ok' \wedge \left(B' \wedge II_C \lhd wait \rhd \left(tr' = tr \wedge a \notin ref' \lhd wait' \rhd tr' = tr ^\frown \langle a \rangle\right)\right)
\end{aligned}
$$

We can now build the graphic form (using Lemma A.1.2):

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$\neg o'$
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	o'
o	$tr' = tr ^\frown \langle a \rangle$	$tr' = tr \wedge a \notin ref'$	$tr' = tr \wedge ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

RHS:

$$
\begin{array}{ll}
 & \exists state, state' \bullet a \rightarrow_X Skip_X \\
= & \text{“ def of a } \rightarrow_X Skip_X \text{ ”} \\
 & \exists state, state' \bullet \mathbf{S}\left(true \vdash \left(\begin{array}{l} tr' = tr \wedge a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \wedge state' = state \end{array} \right) \right) \\
= & \text{“ alternate expansion of } \mathbf{S} \text{ ”} \\
 & \exists state, state' \bullet \\
 & \quad (\exists state' \bullet II_X) \\
 & \quad \lhd wait \rhd \\
 & \quad \left(\left(true \vdash \left(\begin{array}{l} tr' = tr \wedge a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \wedge state' = state \end{array} \right) \right) \wedge tr \leq tr' \right) \\
= & \text{“ push } \exists state, state' \text{ scope inwards ”} \\
 & (\exists state, state' \bullet (\exists state' \bullet II_X)) \\
 & \lhd wait \rhd \\
 & \left(\left(true \vdash \left(\begin{array}{l} tr' = tr \wedge a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \wedge (\exists state, state' \bullet state' = state) \end{array} \right) \right) \wedge tr \leq tr' \right) \\
= & \text{“ merge scopes, 1pt } state' = state \text{ ”} \\
 & (\exists state, state' \bullet II_X) \\
 & \lhd wait \rhd \\
 & \left(true \vdash \left(\begin{array}{l} tr' = tr \wedge a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \wedge True \end{array} \right) \wedge tr \leq tr' \right) \\
= & \text{“ Lemma B.2.2, } \wedge\text{-unit, on page 124 ”} \\
 & II_C \\
 & \lhd wait \rhd \\
 & \left(true \vdash \left(\begin{array}{l} tr' = tr \wedge a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \wedge tr \leq tr' \right) \\
= & \text{“ Lemma B.2.3, on page 124 ”} \\
 & II_C \\
 & \lhd wait \rhd \\
 & \left(DIV \vee ok' \wedge \left(\begin{array}{l} tr' = tr \wedge a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \wedge tr \leq tr' \right)
\end{array}
$$

from previous page:

$$
\begin{array}{ll}
& II_C \\
& \lhd wait \rhd \\
& \left(DIV \lor ok' \land \left(\begin{array}{l} tr' = tr \land a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \land tr \leq tr' \right) \\
= & \text{“ distribute } \land \text{ into } \lhd\rhd \text{ ”} \\
& II_C \\
& \lhd wait \rhd \\
& \left(DIV \lor ok' \land \left(\begin{array}{l} tr' = tr \land a \notin ref' \land tr \leq tr' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \land tr \leq tr' \end{array} \right) \right) \\
= & \text{“ } tr' = tr \text{ and } tr' = tr ^\frown _ \text{ both imply } tr \leq tr' \text{ ”} \\
& II_C \lhd wait \rhd \left(DIV \lor ok' \land \left(\begin{array}{l} tr' = tr \land a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \right) \\
= & \text{“ defn. } II_C \text{ ”} \\
& (DIV \lor ok' \land wait' = wait \land tr' = tr \land ref' = ref) \\
& \lhd wait \rhd \\
& \left(DIV \lor ok' \land \left(\begin{array}{l} tr' = tr \land a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \right) \\
= & \text{“ } (A \lor B \land C) \lhd w \rhd (A \lor B \land D) = A \lor B \land (C \lhd w \rhd D) \text{ (Exercise!) ”} \\
& DIV \lor ok' \land \left(\begin{array}{l} (wait' = wait \land tr' = tr \land ref' = ref) \\ \lhd wait \rhd \\ \left(\begin{array}{l} tr' = tr \land a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \end{array} \right) \\
= & \text{“ } (w' = w \lhd w \rhd \ldots) \equiv (w' \lhd w \rhd \ldots) \text{ ”} \\
& DIV \lor ok' \land \left(\begin{array}{l} (wait' \land tr' = tr \land ref' = ref) \\ \lhd wait \rhd \\ \left(\begin{array}{l} tr' = tr \land a \notin ref' \\ \lhd wait' \rhd \\ tr' = tr ^\frown \langle a \rangle \end{array} \right) \end{array} \right)
\end{array}
$$

We now enter this into the graphical form:

	$\neg w$	$\neg w$	w	w	
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$\neg o'$
$\neg o$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	$tr \leq tr'$	o'
o	$tr' = tr ^\frown \langle a \rangle$	$tr' = tr \land a \notin ref'$	$tr' = tr \land ref' = ref$	.	o'
o	.	.	.	.	$\neg o'$
	$\neg w'$	w'	w'	$\neg w'$	

Both diagrams are identical! ■

B.2 Other Proofs

Theorem B.2.1

$$Stop_C = (\exists state, state' \bullet Stop_X)$$

Proof Strategy: reduce both LHS and RHS to same

LHS:

$$\begin{aligned}
& Stop_C \\
= & \quad \text{“ defn. ”} \\
& \mathbf{R}(true \vdash wait' \wedge tr' = tr) \\
= & \quad \text{“ standard expansion of } \mathbf{R} \text{ ”} \\
& II_C \lhd wait \rhd (true \vdash wait' \wedge tr' = tr) \wedge tr \leq tr'
\end{aligned}$$

This looks like a good target—let's see if we can get the RHS to transform to this.

RHS:

$$\begin{aligned}
& \exists state, state' \bullet Stop_X \\
= & \quad \text{“ defn. ”} \\
& \exists state, state' \bullet \mathbf{S}(true \vdash wait' \wedge tr' = tr) \\
= & \quad \text{“ standard expansion of } \mathbf{S} \text{ ”} \\
& \exists state, state' \bullet (\exists state' \bullet II_X) \lhd wait \rhd (true \vdash wait' \wedge tr' = tr) \wedge tr \leq tr' \\
= & \quad \text{“ } state, state' \text{ not free from } \lhd wait \rhd \text{ rightwards ”} \\
& (\exists state, state' \bullet (\exists state' \bullet II_X)) \lhd wait \rhd (true \vdash wait' \wedge tr' = tr) \wedge tr \leq tr' \\
= & \quad \text{“ remove nested } \exists state' \text{ ”} \\
& (\exists state, state' \bullet II_X) \lhd wait \rhd (true \vdash wait' \wedge tr' = tr) \wedge tr \leq tr' \\
= & \quad \text{“ Lemma B.2.2, on page 124 ”} \\
& II_C \lhd wait \rhd (true \vdash wait' \wedge tr' = tr) \wedge tr \leq tr'
\end{aligned}$$

■

Lemma B.2.2

$$(\exists state, state' \bullet II_X) = II_C$$

Proof Strategy: reduce LHS to RHS

$$\begin{aligned}
& \exists state, state' \bullet II_X \\
= \quad & \text{“ defn. } II_X \text{ ”} \\
& \exists state, state' \bullet \neg ok \wedge tr \leq tr' \\
& \qquad \vee ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge state' = state \\
= \quad & \text{“ restrict quantifier scope ”} \\
& \neg ok \wedge tr \leq tr' \\
& \vee ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge (\exists state, state' \bullet state' = state) \\
= \quad & \text{“ 1-pt rule ”} \\
& \neg ok \wedge tr \leq tr' \vee ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge True \\
= \quad & \text{“ simplify ”} \\
& \neg ok \wedge tr \leq tr' \vee ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \\
= \quad & \text{“ defn } II_C \text{ ”} \\
& II_C
\end{aligned}$$

■

Lemma B.2.3

$$(true \vdash P) \wedge tr \leq tr' = DIV \vee ok' \wedge P \wedge tr \leq tr'$$

Proof Strategy: reduce LHS to RHS

$$\begin{aligned}
& (true \vdash P) \wedge tr \leq tr' \\
= \quad & \text{“ defn. } \vdash \text{ ”} \\
& (ok \Rightarrow ok' \wedge P) \wedge tr \leq tr' \\
= \quad & \text{“ defn. } \Rightarrow \text{ ”} \\
& (\neg ok \vee ok' \wedge P) \wedge tr \leq tr' \\
= \quad & \text{“ distributivity ”} \\
& \neg ok \wedge tr \leq tr' \vee ok' \wedge P \wedge tr \leq tr' \\
= \quad & \text{“ defn. } DIV \text{ ”} \\
& DIV \vee ok' \wedge P \wedge tr \leq tr'
\end{aligned}$$

■

Theorem B.2.4

$$\begin{array}{rcl} A_C \,;_C\, B_C & \sqsubseteq & (\exists state, state' \bullet A_X \,;_X\, B_X) \\ \textbf{\textit{where}} & & A_C = \exists state, state' \bullet A_X \\ & & B_C = \exists state, state' \bullet B_X \end{array}$$

This proof is based on distinguishing clearly between the reactive and state observations. We introduce the following shorthands: $R = \{ok, wait, tr, ref\}$, s and s' for *state* and *state*$'$, and note the following alternate formulations of both kinds of sequential composition:

$$\begin{array}{rcl} P \,;_C\, Q & = & \exists R_m \bullet P[R_m/R'] \wedge Q[R_m/R] \\ P \,;_X\, Q & = & \exists R_m, s_m \bullet P[R_m, s_m/R', s'] \wedge Q[R_m, s_m/R, s] \end{array}$$

Proof (with Andrew Butterfield)
Strategy: we show LHS refined by RHS by starting with the RHS and proving it implies the LHS, as per the definition of refinememt.

$$\begin{array}{ll}
& RHS = (\exists s, s' \bullet A_X \,;_X\, B_X) \\
= & \quad \text{“ alt. defn. of } ;_X \text{ ”} \\
& \exists s, s' \bullet (\exists R_m, s_m \bullet A_X[R_m, s_m/R', s'] \wedge B_X[R_m, s_m/R, s]) \\
= & \quad \text{“ re-order and re-nest quantifiers ”} \\
& \exists R_m \bullet (\exists s_m, s, s' \bullet A_X[R_m, s_m/R', s'] \wedge B_X[R_m, s_m/R, s]) \\
\Rightarrow & \quad \text{“ weakening ”} \\
& \exists R_m \bullet (\exists s_m, s, s' \bullet A_X[R_m, s_m/R', s']) \wedge (\exists s_m, s, s' \bullet B_X[R_m, s_m/R, s]) \\
= & \quad \text{“ } s' \text{ not free in 1st conjunct, nor } s \text{ in 2nd ”} \\
& \exists R_m \bullet (\exists s_m, s \bullet A_X[R_m, s_m/R', s']) \wedge (\exists s_m, s' \bullet B_X[R_m, s_m/R, s]) \\
= & \quad \text{“ independent substitutions ”} \\
& \exists R_m \bullet (\exists s_m, s \bullet A_X[s_m/s'][R_m/R']) \wedge (\exists s_m, s' \bullet B_X[s_m/s][R_m/R]) \\
= & \quad \text{“ move } R_m \text{ substitution out ”} \\
& \exists R_m \bullet (\exists s_m, s \bullet A_X[s_m/s'])[R_m/R'] \wedge (\exists s_m, s' \bullet B_X[s_m/s])[R_m/R] \\
= & \quad \text{“ defn } ;_C \text{ ”} \\
& (\exists s_m, s \bullet A_X[s_m/s']) \,;_C\, (\exists s_m, s' \bullet B_X[s_m/s]) \\
= & \quad \text{“ } \alpha\text{-renaming ”} \\
& (\exists s', s \bullet A_X) \,;_C\, (\exists s, s' \bullet B_X)
\end{array}$$

■

Theorem B.2.5

$$A_C \,{}_9^o{}_C\, J_C = (\exists\, state, state' \bullet A_X \,{}_9^o{}_X\, J_X)$$

Proof (with Andrew Butterfield)

Strategy: Making R.H.S equal to L.H.S.

Here we note that

$$
\begin{array}{lcl}
Obs^C & = & ok, wait, tr, ref \\
Obs^X & = & ok, wait, tr, ref, state \\
Obs^C & = & Obs^X \setminus_{state} \\
RHS: & & \\
 & & \exists\, state, state' \bullet A_X \,{}_9^o{}_X\, J_X \\
 & = & \text{“ def of } {}_9^o{}_X \text{ ”} \\
 & & \exists\, state, state' \bullet (\exists\, Obs^X_m \bullet A_X[Obs^X_m/Obs'^X] \wedge J_X[Obs^X_m/Obs^X]) \\
 & = & \text{“ flatten nested } \exists \text{ ”} \\
 & & \exists\, state, state', Obs^X_m \bullet A_X[Obs^X_m/Obs'^X] \wedge J_X[Obs^X_m/Obs^X] \\
 & = & \text{“ def of } J_X \text{ ”} \\
 & & \exists\, state, state', Obs^X_m \bullet A_X[Obs^X_m/Obs'^X] \\
 & & \quad \wedge ((ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge state' = state)[Obs^X_m/Obs^X] \\
 & = & \text{“ substitution ”} \\
 & & \exists\, state, state', Obs^X_m \bullet A_X[Obs^X_m/Obs'^X] \\
 & & \quad \wedge (ok_m \Rightarrow ok') \wedge wait' = wait_m \wedge tr' = tr_m \wedge ref' = ref_m \wedge state' = state_m \\
 & = & \text{“ one-pt rule, replacing } state' \text{ only ”} \\
 & & \exists\, state, Obs^X_m \bullet A_X[Obs^X_m/Obs'^X][state_m/state'] \\
 & & \quad \wedge (ok_m \Rightarrow ok') \wedge wait' = wait_m \wedge tr' = tr_m \wedge ref' = ref_m \\
 & = & \text{“ All } state' \text{ in } A_X \text{ are now } state_m\text{, so outer sub has no effect ”} \\
 & & \exists\, state, Obs^X_m \bullet A_X[Obs^X_m/Obs'^X] \\
 & & \quad \wedge (ok_m \Rightarrow ok') \wedge wait' = wait_m \wedge tr' = tr_m \wedge ref' = ref_m \\
 & = & \text{“ } state \text{ only occurs, if at all, in } A_X \text{ so we can distr } \exists state \text{ in ”} \\
 & & \exists\, Obs^X_m \bullet (\exists\, state \bullet A_X)[Obs^X_m/Obs'^X] \\
 & & \quad \wedge (ok_m \Rightarrow ok') \wedge wait' = wait_m \wedge tr' = tr_m \wedge ref' = ref_m \\
 & = & \text{“ substitution backwards ”} \\
 & & \exists\, Obs^X_m \bullet (\exists\, state \bullet A_X)[Obs^X_m/Obs'^X] \\
 & & \quad \wedge ((ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref)[Obs^X_m/Obs^X] \\
 & = & \text{“ def } J_C \text{ ”} \\
 & & \exists\, Obs^X_m \bullet (\exists\, state \bullet A_X)[Obs^X_m/Obs'^X] \wedge J_C[Obs^X_m/Obs^X] \\
 & = & \text{“ } J_C \text{ does not mention } state \text{ or } state' \text{ so we can restrict to } Obs^C \text{ ”} \\
 & & \exists\, Obs^X_m \bullet (\exists\, state \bullet A_X)[Obs^X_m/Obs'^X] \wedge J_C[Obs^C_m/Obs^C] \\
 & = & \text{“ factor } state_m \text{ out of } Obs^X_m \text{ ”} \\
 & & \exists\, state_m, Obs^C_m \bullet (\exists\, state \bullet A_X)[state_m, Obs^C_m/state', Obs'^C] \wedge J_C[Obs^C_m/Obs^C] \\
 & = & \text{“ can split substitution because } state_m \text{ is not in } A_X\text{. ”} \\
 & & \exists\, state_m, Obs^C_m \bullet (\exists\, state \bullet A_X)[Obs^C_m/Obs'^C][state_m,/state'] \wedge J_C[Obs^C_m/Obs^C]
\end{array}
$$

$=$ " can bring $state'$ substitution out, as only poss. occurrences are in A_X "

$\exists state_m, Obs^C_m \bullet ((\exists state \bullet A_X)[Obs^C_m/Obs'^C] \wedge J_C[Obs^C_m/Obs^C])[state_m,/state']$

$=$ " $(\exists x \bullet P) = (\exists y \bullet P[y/x]$, provided $y \notin x, P$ "

$\exists state', Obs^C_m \bullet ((\exists state \bullet A_X)[Obs^C_m/Obs'^C] \wedge J_C[Obs^C_m/Obs^C])$

$=$ " $state'$ only in A_X, if at all "

$\exists Obs^C_m \bullet ((\exists state, state' \bullet A_X)[Obs^C_m/Obs'^C] \wedge J_C[Obs^C_m/Obs^C])$

$=$ " def $\fatsemi_C$ "

$(\exists state, state' \bullet A_X) \fatsemi_C J_C$

$=$ " def A_X "

$A_C \fatsemi_C J_C$

LHS!

■

Theorem B.2.6

$$\mathbf{Rn}(A_C) \sqsubseteq \exists state, state' \bullet \mathbf{Sn}(A_X), \quad \boldsymbol{n \in 1 \ldots 3}$$

Proof For n=1,

$$
\begin{array}{cl}
 & L.H.S = \mathbf{R1}(A_C) \\
= & \quad \text{“ def of } \mathbf{R1} \text{ ”} \\
 & A_C \wedge tr \leq tr' \\
\sqsubseteq & \quad \text{“ def of A}_C \text{ ”} \\
 & \exists state, state' \bullet A_X \wedge tr \leq tr' \\
= & \quad \text{“ def of } \mathbf{S1} \text{ ”} \\
 & \exists state, state' \bullet \mathbf{S1}(A_X) = R.H.S
\end{array}
$$

For n=2,

$$
\begin{array}{cl}
 & \mathbf{R2}(A_C) \\
= & \quad \text{“ def of } \mathbf{R2} \text{ ”} \\
 & \exists s \bullet A_C[s, s \frown (tr' - tr)/tr, tr'] \\
\sqsubseteq & \quad \text{“ def of A}_C \text{ ”} \\
 & \exists s \bullet \exists state, state' \bullet A_X[s, s \frown (tr' - tr)/tr, tr'] \\
= & \quad \text{“ Re-arranging ”} \\
 & \exists state, state' \bullet \exists s \bullet A_X[s, s \frown (tr' - tr)/tr, tr'] \\
= & \quad \text{“ def of } \mathbf{S2} \text{ ”} \\
 & \exists state, state' \bullet \mathbf{S2}(A_X) = R.H.S
\end{array}
$$

For n=3,

$$
\begin{array}{cl}
 & \mathbf{R3}(A_C) \\
= & \quad \text{“ def of } \mathbf{R3} \text{ ”} \\
 & I\!I \lhd wait \rhd A_C \\
= & \quad \text{“ def of } \lhd \rhd \text{ ”} \\
 & wait \wedge I\!I \vee \neg wait \wedge A_C \\
\sqsubseteq & \quad \text{“ def of A}_C \text{ ”} \\
 & (wait \wedge I\!I) \vee \neg wait \wedge (\exists state, state' \bullet A_X) \\
= & \quad \text{“ ”} \\
 & \exists state, state' \bullet I\!I \wedge wait \vee \neg wait \wedge A_X \\
= & \quad \text{“ def of } \lhd \rhd \text{ ”} \\
 & \exists state, state' \bullet \exists state' \bullet I\!I \lhd wait \rhd A_X \\
= & \quad \text{“ def of } \mathbf{S3} \text{ ”} \\
 & \exists state, state' \bullet \mathbf{S3}(A_X) = L.H.S
\end{array}
$$

■

Theorem B.2.7

$$\mathbf{CSPn}(A_C) \sqsubseteq \exists state, state' \bullet \mathbf{CXSn}(A_X), \quad \boldsymbol{n} \in \boldsymbol{1} \ldots \boldsymbol{5}$$

Proof For n=1,

$$
\begin{aligned}
& L.H.S = \mathbf{CSP1}(A_C) \\
= & \quad \text{“ def of } \mathbf{CSP1} \text{ ”} \\
& A_C \vee DIV \\
\sqsubseteq & \quad \text{“ def of } A_C \text{ ”} \\
& \exists state, state' \bullet A_X \vee DIV \\
= & \quad \text{“ def of } \mathbf{CXS1} \text{ ”} \\
& \exists state, state' \bullet \mathbf{CXS1}(A_X) = R.H.S
\end{aligned}
$$

For n=2,

$$
\begin{aligned}
& L.H.S = \mathbf{CSP2}(A_C) \\
= & \quad \text{“ def of } \mathbf{CSP2} \text{ ”} \\
& A_C \,{}_{9}^{\circ}{}_C\, J_C \\
= & \quad \text{“ By theorem B.2.5, on page 126 ”} \\
& \exists state, state' \bullet A_X \,{}_{9}^{\circ}{}_X\, J_X \\
= & \quad \text{“ def of } \mathbf{CXS2} \text{ ”} \\
& \exists state, state' \bullet \mathbf{CXS2}(A_X) = R.H.S
\end{aligned}
$$

For n=4,

$$
\begin{aligned}
& L.H.S = \mathbf{CSP4}(A_C) \\
= & \quad \text{“ def of } \mathbf{CSP4} \text{ ”} \\
& A_C \,{}_{9}^{\circ}{}_C\, Skip_C \\
\sqsubseteq & \quad \text{“ def of } A_C \text{ ”} \\
& (\exists state, state' \bullet A_X) \,{}_{9}^{\circ}{}_C\, Skip_C \\
= & \quad \text{“ By theorem 6.2.1, on page 49 ”} \\
& (\exists state, state' \bullet A_X) \,{}_{9}^{\circ}{}_C\, (\exists state, state' \bullet Skip_X) \\
= & \quad \text{“ By theorem B.2.4, on page 125 ”} \\
& \exists state, state' \bullet (A_X \,{}_{9}^{\circ}{}_X\, Skip_X) \\
= & \quad \text{“ def of } \mathbf{CXS4} \text{ ”} \\
& \exists state, state' \bullet \mathbf{CXS4}(A_X) = R.H.S
\end{aligned}
$$

■

B.3 Lemmas

Proofs of following two lemmas have been gone through and can be provided if needed.

Lemma 1

Lemma B.3.1 *Assuming that A and B are* **R***-healthy:*

$$\mathbf{R}\left(\begin{array}{l} \exists Obs_A, Obs_B \bullet \\ \quad A[Obs_A/Obs'] \wedge B[Obs_B/Obs'] \\ \quad \wedge\ ok' = ok_A \wedge ok_B \\ \quad \wedge\ wait' = wait_A \vee wait_B \\ \quad \wedge\ ref' \subseteq (ref_A \cup ref_B) \cap S \cup (ref_A \cap ref_B) \setminus S \\ \quad \wedge\ tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \end{array}\right) = \left(\begin{array}{l} \exists Obs_A, Obs_B \bullet \\ \quad A[Obs_A/Obs'] \wedge B[Obs_B/Obs'] \\ \quad \wedge\ ok' = ok_A \wedge ok_B \\ \quad \wedge\ wait' = wait_A \vee wait_B \\ \quad \wedge\ ref' \subseteq (ref_A \cup ref_B) \cap S \cup (ref_A \cap ref_B) \setminus S \\ \quad \wedge\ tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \end{array}\right)$$

Lemma 2

Lemma B.3.2 *Assuming that A and B are* **R***-healthy:*

$$\left(\begin{array}{l} \exists Obs_A, Obs_B \bullet \\ \quad A[Obs_A/Obs'] \wedge B[Obs_B/Obs'] \\ \quad \wedge\ ok' = ok_A \wedge ok_B \\ \quad \wedge\ wait' = wait_A \vee wait_B \\ \quad \wedge\ ref' \subseteq (ref_A \cup ref_B) \cap S \cup (ref_A \cap ref_B) \setminus S \\ \quad \wedge\ tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \end{array}\right) ; Skip$$

$$= \left(\begin{array}{l} \exists Obs_A, Obs_B \bullet \\ \quad A[Obs_A/Obs'] \wedge B[Obs_B/Obs'] \\ \quad \wedge\ ok' = ok_A \wedge ok_B \\ \quad \wedge\ wait' = wait_A \vee wait_B \\ \quad \wedge\ tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \\ \quad \wedge \left(\begin{array}{c} ref' \subseteq (ref_A \cup ref_B) \cap S \cup (ref_A \cap ref_B) \setminus S \\ \lhd wait' \rhd \\ true \end{array}\right) \end{array}\right)$$

Lemma 3

Lemma B.3.3

$$tr \leq tr_A \wedge tr \leq tr_B \wedge tr' - tr \in (tr_A - tr) \parallel_S (tr_B - tr) \Rightarrow tr \leq tr'$$

Appendix C

Haskell Implementation

C.1 Working with Simple Circus in Haskell

```
module SimpleCircus where
import Data.List
import Maybe
import Utilities
```

C.1.1 Names

```
type Name = String
```

C.1.2 Expressions

```
type EvtSpec = (Name, [Comm])

data Comm
 = Null | Dot Expr | Bang Expr | Q Name (Maybe Expr)
 deriving (Eq, Ord, Show)

data Expr
 = B Bool | Z Int | Var Name | Val Name
 | Agg Name Name [Expr]
 | App Name [Expr]
 | Bin Name Expr Expr
 | Evts EvtSpec
 deriving (Eq,Ord,Show)
```

C.1.3 Abstract Syntax

```
data Circus
 = Div | Stop | Skip |
 Chaos Expr |
 String :-> Circus |
 (String,[String]) ::-> Circus |
 Expr :::-> Circus | -- to accomodate I/O channels
 Circus ::: Circus |
 IntChoice Circus Circus |
 ExtChoice Circus Circus |
 Hide Circus [String] |
 IPar [String] Circus Circus |
 APar [String] [String] Circus Circus |
 Ilv Circus Circus |
 Cond Expr Circus Circus |
 Guard Expr Circus |
 INT Name [String] Circus |
 SEQ Name [String] Circus |
 EXT Name [String] Circus |
 ILV Name [String] Circus |
 IPAR [String] Name [String] Circus |
 APAR Name [String] [String] Circus |
 Name := Expr |
 Call String [Expr] |   --  name(actual arguments,)
 Mu Name Circus
 deriving (Eq,Ord,Show)

type CircusDef  --  name(formal arguments,) = body
   = ( String     -- name
     , ( [String]   -- formal argument
       , Circus     -- body
       )
     )

type CircusProgram = [CircusDef] -- ordered by name component
```

C.1.4 Simplifying Constructors

```
(x,[]) --> circ  =  Stop
(x,xs) --> circ  =  (x,xs) ::-> circ

mkIntChoice Div circ = Div
mkIntChoice circ Div = Div
mkIntChoice c1 c2 = IntChoice c1 c2

mkINT nm [] _ = Div
mkINT nm [x] circ = csubstitute x nm circ
mkINT nm [x,y] circ = IntChoice (csubstitute x nm circ) (csubstitute y nm circ)
mkINT nm es circ = INT nm es circ

mkExtChoice circ Div = Div
mkExtChoice Div circ = Div
mkExtChoice Stop circ = circ
mkExtChoice circ Stop = circ
mkExtChoice c1 c2 = ExtChoice c1 c2

mkEXT nm [] _ = Div
mkEXT nm [x] circ = csubstitute x nm circ
mkEXT nm [x,y] circ = ExtChoice (csubstitute x nm circ) (csubstitute y nm circ)
mkEXT nm es circ = EXT nm es circ
```

Substitutions

```
csubstitute e x pr@((y,es) ::-> p)
  | x == y  =  pr
  | otherwise  =  (y,map (esubstitute e x ) es) ::-> (csubstitute e x p)

csubstitute e x (IntChoice pr1 pr2) = IntChoice (csubstitute e x pr1)
                                         (csubstitute e x pr2)

csubstitute e x circ = circ

esubstitute e x y
 | x == y     =  e
 | otherwise  =  y
```

C.2 Standard Circus Names

```
module StdCircusNames where
import SimpleCircus
```

C.2.1 Names

Names for Binary Operators

```
notName = " not "
orName = " or "
andName = " and "
eqvName = " ^= "
impName = " ==> "
```

Names of Operators for Comparisons

```
eqName = " = "
neqName = " != "
ltName = " < "
leName = " <= "
gtName = " > "
geName = " >= "
```

Arithmetic Operators

```
addName = "+"
subName = "-"
mulName = "*"
divName = "/"
remName = "%"
absName = "abs"
```

Names of Set Notations

```
setOpenName = "{"
setCloseName = "}"
chanSetOpenName = "{|"
```

```
chanSetCloseName = "|}"
unionName = "union"
intersectName = "intersct"
setdiffName = "diff"
subsetName = "subset"
```

Names for Sequences

```
seqOpenName = "<"
seqCloseName = ">"
catName = "^"
lenName = "#"
```

Circus Specific Names

```
ilvName = "|||"
interfaceParOpenName = "[|"
interfaceParCloseName = "|]"
alphabetisedParOpenName = "["
alphabetisedParMidName = "||"
alphabetisedParCloseName = "]"
repAlphabetisedOpenName = "["
repAlphabetisedCloseName = "]"
icName = "|~|"
ecName = "[]"
seqName = ";"
guardName = "&"
prefixName = "->"
hideName = "\\"
atName = "@"
replName = "RR"
ifName = "if"
thenName = "then"
elseName = "else"
asgName = ":="
muName = "Mu "
```

C.2.2 Expression Builders

```
true = B True
false = B False

land = Bin andName
```

```
lor = Bin orName
implies = Bin impName
eqv = Bin eqvName
lnot = App notName

eq = Bin eqName
neq = Bin neqName
lt = Bin ltName
le = Bin leName
gt = Bin gtName
ge = Bin geName
subset = Bin subsetName
setof = Agg setOpenName setCloseName
setnull = setof []
unn = Bin unionName
intsct = Bin intersectName
setdiff = Bin setdiffName
```

C.2.3 Precedence

```
stdPrec nm
  | nm == eqvName = 21
  | nm == impName = 22
  | nm == orName = 23
  | nm == andName = 24
  | nm == eqName = 25
  | nm == neqName = 26
  | nm == ltName = 26
  | nm == leName = 26
  | nm == gtName = 26
  | nm == geName = 26
  | nm == addName = 27
  | nm == subName = 27
  | nm == mulName = 28
  | nm == divName = 28
  | nm == remName = 28
  | nm == catName = 29
  | otherwise = 1
```

```
cspPrec nm
  | nm == ilvName = 2
  | nm == interfaceParOpenName = 3
  | nm == alphabetisedParMidName = 3
  | nm == icName = 4
  | nm == ecName = 5
  | nm == seqName = 8
  | nm == guardName = 9
```

```
  | nm == prefixName = 10
  | nm == hideName = 11
  | nm == muName = 12
  | nm == ifName = 1
  | nm == replName = 1
  | otherwise = 1
```

C.3 Standard Circus Printing

```
module StdCircusPrint where
import Data.List
import Maybe
import Utilities
import SimpleCircus
import StdCircusNames
```

C.3.1 Pretty-Printing Expressions

```
type Precf = Name -> Int

ppExpr :: Precf -> Expr -> String

ppExpr prec e
  = pp 0 e
  where
   pp _ (B False) = "ff"
   pp _ (B True) = "tt"
   pp _ (Z z) | z < 0 = brkt $ show z | otherwise = show z
   pp _ (Var v) = v
   pp _ (Val v) = v
   pp _ (App nm es) = nm ++ brkt (showlist (pp 0) es)
   pp _ (Agg open close es) = open ++ showlist (pp 0) es ++ close
   pp _ (Evts (ch, rest)) = ch ++ concat (map (ppComm stdPrec) rest)
   pp cp (Bin nm e1 e2)
     = precRender prec cp nm
    (\nmp -> pp nmp e1 ++ nm ++ pp nmp e2)

   --pp prec _ = ""
   --pp prec (c:cs) = pp prec c ++ pp prec cs

pad s = ' ':s++" "

brkt s = "(" ++ s ++ ")"

precRender prec cp nm strf
 | nmp > cp = string
 | otherwise = brkt string
 where
   nmp = prec nm
   string = strf nmp

showlist sh   [] = ""
showlist sh   [x] = sh x
showlist sh (x:xs) = sh x ++ "," ++ showlist sh xs
```

```
-- Pretty printing of (String, [String])
ppSts (x,as) = x ++ ":" ++ ppSet as
ppSet as = "{"++ppSts1 as++"}"
-- Pretty printing of [String]
ppSts1 = concat . intersperse ","
```

Pretty Printing Events

```
ppComm :: Precf -> Comm -> String
ppComm prec Null = ""
ppComm prec (Dot expr) = "." ++ (ppExpr prec expr)
ppComm prec (Bang expr) = "!" ++ (ppExpr prec expr)
ppComm prec (Q nm Nothing) = "?" ++ nm
ppComm prec (Q nm (Just expr)) = "?" ++ nm ++ ":" ++ (ppExpr prec expr)

ppComms prec [] = ""
ppComms prec (c:cs) = ppComm prec c ++ ppComms prec cs
```

C.3.2 Pretty-Printing Processes

```
ppProc :: Precf -> Circus -> String
ppProc prec e
  = pp 0 e
  where
    pp _ Div = "Div"
    pp _ Stop = "Stop"
    pp _ Skip = "Skip"
    pp _ (Chaos a) = "Chaos(" ++ ppExpr prec a ++ ")"

    pp cp (a :-> circ) -- Prefix action
     = precRender prec cp prefixName
       (\nmp -> a
            ++ pad prefixName
++ pp nmp circ)

    pp cp ((x,xs) ::-> circ) -- Prefix action
     = precRender prec cp prefixName
       (\nmp -> ppSts (x,xs)
            ++ pad prefixName
++ pp nmp circ)

    pp cp (evtExpr :::-> circ) -- I/O channel
      = precRender prec cp prefixName
       (\nmp -> ppExpr prec evtExpr
             ++ pad prefixName
 ++ "\n\t"
             ++ pp nmp circ)

    pp cp (c1 ::: c2) -- Seq composition i.e c1 ; c2
```

```
      = precRender prec cp seqName
        (\nmp -> pp nmp c1 ++ pad seqName ++ "\n\t" ++ pp nmp c2)

    pp cp (Hide circ a) -- P \ A
      = precRender prec cp hideName
        (\nmp -> pp nmp circ ++ pad hideName ++ ppSet a)

    pp cp (ExtChoice c1 c2) -- p [] q
      = precRender prec cp ecName
        (\nmp -> pp nmp c1 ++ "\n\t" ++ pad ecName ++ "\n\t" ++ pp nmp c2)

    pp cp (IntChoice c1 c2) -- p |~| q
      = precRender prec cp icName
        (\nmp -> pp nmp c1 ++ pad icName ++ pp nmp c2)

    pp cp (Guard b circ) -- b & P
      = precRender prec cp guardName
        (\nmp -> ppExpr prec b ++ pad guardName ++ "\n\t" ++ pp nmp circ)

    pp cp (Ilv c1 c2) -- p ||| q
      = precRender prec cp ilvName
        (\nmp -> pp nmp c1 ++ pad ilvName ++ pp nmp c2)

    pp cp (IPar a c1 c2) -- p [| a |] q
      = precRender prec cp interfaceParOpenName
        (\nmp -> pp nmp c1
             ++ ' ':interfaceParOpenName
++ ' ':(ppSts1 a)
++ ' ':interfaceParCloseName
++ ' ':pp nmp c2)

    pp cp (APar a b c1 c2)  -- p [a || a'] q
      = precRender prec cp alphabetisedParOpenName
        (\nmp -> pp nmp c1
             ++ ' ':alphabetisedParOpenName
++ ' ':(ppSts1 a)
++ ' ':alphabetisedParMidName
++ ' ':(ppSts1 b)
++ ' ':alphabetisedParCloseName
++ ' ':pp nmp c2)

    pp cp (SEQ x s circ) -- ; x:s @ p
      = precRender prec cp replName
        (\nmp -> seqName
             ++ ' ':x ++ ":<" ++ (ppSts1 s)
++ "> @ "
++ pp nmp circ)

    pp cp (EXT x a circ) -- [] x:a @ p
      = precRender prec cp replName
        (\nmp -> ecName
             ++ ' ':x ++ ":" ++ (ppSet a)
++ " @ "
```

```
++ pp nmp circ)

    pp cp (INT x a circ) -- |~| x:a @ p
      = precRender prec cp replName
        (\nmp -> icName
             ++ ' ':x ++ ":" ++ (ppSet a)
++ " @ "
++ pp nmp circ)

    pp cp (ILV x a circ) -- ||| x:a @ p
      = precRender prec cp replName
        (\nmp -> ilvName
             ++ ' ':x ++ ":" ++ (ppSet a)
++ " @ "
++ pp nmp circ)

    pp cp (IPAR a' x a circ)  -- [| a' |] x:a @ p
      = precRender prec cp replName
        (\nmp -> interfaceParOpenName
             ++ (ppSet a')
++ interfaceParCloseName
++ ' ':x ++ ":" ++ (ppSet a)
++ " @ "
++ pp nmp circ)

    pp cp (APAR x a a' circ)  -- || x:a @ [a'] p
      = precRender prec cp replName
        (\nmp -> alphabetisedParMidName
             ++ ' ':x ++ ":" ++ (ppSet a)
++ " @ "
++ repAlphabetisedOpenName
++ (ppSet a')
++ repAlphabetisedCloseName
++ ' ':pp nmp circ)

    pp cp (Cond b c1 c2) -- if b then p else q
      = precRender prec cp ifName
        (\nmp -> ifName
             ++ ' ':ppExpr prec b
++ "\n\t"
++ ' ':thenName
++ ' ': pp nmp c1
++ "\n\t"
++ ' ':elseName
++ ' ':pp nmp c2)

    pp cp (n := e)
     = precRender prec cp asgName
        (\nmp -> n ++ pad asgName ++ ppExpr prec e)

    pp cp (Call f []) = f
    pp cp (Call f es)
     = f ++ "("++concat(intersperse "," (map (ppExpr prec) es))++")"
```

```
    pp cp (Mu nm circ) = precRender prec cp muName
   (\nmp -> muName ++ nm ++ " @ " ++ pp nmp circ ++ " ; " ++ nm)

    --pp cp circ = "(CAN'T PP : "++show circ++")"

pp = ppProc  stdPrec

putpp = putStrLn . pp
```

Pretty-printing definitions

```
ppDefs prec = unlines . map (ppDef prec)

ppDef prec (name, ([],body) )
 = name ++ " =^= " ++ ppProc prec body ++ "\n"
ppDef prec (name, (fpars,body) )
 = name ++ "(" ++ ppSts1 fpars ++")" ++ " =^= " ++ ppProc prec body ++ "\n"

ppd = ppDefs stdPrec

putppd = putStrLn . ppd
```

C.4 Simple Circus Translation

```
module CircusNameMgmt where
import Data.List
import Maybe
import Utilities
import SimpleCircus
```

C.4.1 Name Management

We need to determine all the names in a Circus expression

```
circNames :: Circus -> [String]
circNames Div = []
circNames Stop = []
circNames Skip = []
circNames (Chaos e) = []
circNames (a :-> circ) = lnorm (a:(circNames circ))
circNames ((x,xs)::-> circ) = lnorm (x:xs ++ (circNames circ))
circNames (c1 ::: c2) = lnorm ((circNames c1) ++ (circNames c2))
circNames (IntChoice c1 c2) = lnorm ((circNames c1) ++ (circNames c2))
circNames (ExtChoice c1 c2) = lnorm ((circNames c1) ++ (circNames c2))
circNames (Hide circ a) = lnorm ((circNames circ) ++ a)
circNames (IPar x c1 c2) = lnorm (x ++ (circNames c1) ++ (circNames c2))
circNames (APar x y c1 c2) = lnorm (x ++ y ++ (circNames c1) ++ (circNames c2))
circNames (Ilv c1 c2) = lnorm ((circNames c1) ++ (circNames c2))
circNames (Cond e c1 c2) = lnorm ((circNames c1) ++ (circNames c2))
circNames (Guard e circ) = lnorm (circNames circ)
circNames (INT nm a circ) = lnorm (nm:a ++ (circNames circ))
circNames (SEQ nm a circ) = lnorm (nm:a ++ (circNames circ))
circNames (EXT nm a circ) = lnorm (nm:a ++ (circNames circ))
circNames (ILV nm a circ) = lnorm (nm:a ++ (circNames circ))
circNames (IPAR x nm y circ) = lnorm (nm:x ++ y ++ (circNames circ))
circNames (APAR nm x y circ) = lnorm (nm:x ++ y ++ (circNames circ))
circNames (nm := e) = lnorm (nm:[])
circNames (Call a exprs) = lnorm (a:[])
```

We need to generate a new name that is not present in a given list

```
freshName :: [String] -> String
freshName known = isFirstUnknown known allNames

isFirstUnknown known (n:ns)
  | n `elem` known = isFirstUnknown known ns
  | otherwise = n

allNames = genName 'a' 0

genName cseed nseed
  = [cseed,intToDigit nseed]:genName cseed' nseed'
```

```
   where
      cseed' = if cseed == 'z' then 'a' else chr (ord cseed+1)
      nseed' = if cseed == 'z' then (nseed+1) else nseed

ord  :: Char -> Int
ord  =  fromEnum

chr  :: Int  -> Char
chr  =  toEnum

intToDigit :: Int -> Char
intToDigit i
  | i >= 0  && i <=  9   =  toEnum (fromEnum '0' + i)
  | i >= 10 && i <= 15   =  toEnum (fromEnum 'a' + i - 10)
  | otherwise            =  error "Char.intToDigit: not a digit"
```

```
freshNameAct :: [String] -> String
freshNameAct known = isFirstUnknown1 known allNames1

isFirstUnknown1 known (n:ns)
   | n `elem` known = isFirstUnknown1 known ns
   | otherwise = n

allNames1 = genName1 'A' 0

genName1 cseed nseed
   = [cseed,intToDigit1 nseed]:genName1 cseed' nseed'
   where
      cseed' = if cseed == 'Z' then 'A' else chr (ord cseed+1)
      nseed' = if cseed == 'Z' then (nseed+1) else nseed

intToDigit1 :: Int -> Char
intToDigit1 i
  | i >= 0  && i <=  9   =  toEnum (fromEnum '0' + i)
  | i >= 10 && i <= 15   =  toEnum (fromEnum 'A' + i - 10)
  | otherwise            =  error "Char.intToDigit: not a digit"
```

C.5 Simple Circus Laws

```
module SimpleCircusLaws where
import Data.List
import Maybe
import Utilities
import SimpleCircus
import StdCircusNames
import CircusNameMgmt
```

C.5.1 Laws

“; -assoc” $P;\ (Q;\ R) = (P;\ Q);\ R$

```
law_Seq_Assoc_LtoR (c1 ::: (c2 ::: c3)) = (True, ((c1 ::: c2) ::: c3))
law_Seq_Assoc_LtoR c = (False, c)

law_Seq_Assoc_RtoL ((c1 ::: c2) ::: c3) = (True, (c1 ::: (c2 ::: c3)))
law_Seq_Assoc_RtoL c = (False, c)
```

“; -unit-l” $SKIP \fatsemi P = P$

```
law_Skip_Unit_LtoR (Skip ::: c) = (True,c)
law_Skip_Unit_LtoR c = (False, c)
```

“; -unit-r” $P \fatsemi SKIP = P$

```
law_Skip_Unit_RtoL (c ::: Skip) = (True,c)
law_Skip_Unit_RtoL c = (False, c)
```

“; -zero-l” $STOP \fatsemi P = STOP$

```
law_Stop_Zero_LtoR (Stop ::: _) =  (True, Stop)
law_Stop_Zero_LtoR c = (False, c)
```

“prefix-assoc-l” $(a \rightarrow P_1);\ P_2 = a \rightarrow (P_1;\ P_2)$

```
law_Prefix_Assoc_LtoR ((a :-> c1 ) ::: c2 ) = (True, a :-> (c1 ::: c2))
law_Prefix_Assoc_LtoR c = (False, c)
```

“⊓-Idem” $P \sqcap P = P$

```
law_IntChoice_IdemL c@(IntChoice c1 c2)
  | c1 == c2  = (True, c1)
  | otherwise  =  (False, c)
law_IntChoice_IdemL c = (False, c)
law_IntChoice_IdemR c = (True, IntChoice c c)
```

“□-Idem” $P \Box P = P$

```
law_ExtChoice_IdemL c@(ExtChoice c1 c2)
  | c1 == c2  = (True, c1)
  | otherwise  =  (False, c)
law_ExtChoice_IdemL c = (False, c)
law_ExtChoice_IdemR c = (True, ExtChoice c c)
```

“⊓-Symm” $P_1 \sqcap P_2 = P_2 \sqcap P_1$

```
law_IntChoice_SymmL (IntChoice c1 c2) = (True,(IntChoice c2 c1))
law_IntChoice_SymmL c = (False, c)
law_IntChoice_SymmR = law_IntChoice_SymmL
```

“□-Symm” $P_1 \Box P_2 = P_2 \Box P_1$

```
law_ExtChoice_SymmL (ExtChoice c1 c2) = (True,(ExtChoice c2 c1))
law_ExtChoice_SymmL c = (False, c)
law_ExtChoice_SymmR = law_ExtChoice_SymmL
```

“⊓-assoc-l” $P_1 \sqcap (P_2 \sqcap P_3) = (P_1 \sqcap P_2) \sqcap P_3$

```
law_IntChoice_Assoc_LtoR (IntChoice c1 (IntChoice c2 c3))
        = (True,(IntChoice (IntChoice c1 c2) c3))
law_IntChoice_Assoc_LtoR c = (False,c)
law_IntChoice_Assoc_RtoL (IntChoice (IntChoice c1 c2) c3)
        = (True, (IntChoice c1 (IntChoice c2 c3)))
law_IntChoice_Assoc_RtoL c = (False, c)
```

“□-assoc-l” $P_1 \Box (P_2 \Box P_3) = (P_1 \Box P_2) \Box P_3$

```
law_ExtChoice_Assoc_LtoR (ExtChoice c1 (ExtChoice c2 c3))
        = (True,(ExtChoice (ExtChoice c1 c2) c3))
law_ExtChoice_Assoc_LtoR c = (False,c)
law_ExtChoice_Assoc_RtoL (ExtChoice (ExtChoice c1 c2) c3)
        = (True, (ExtChoice c1 (ExtChoice c2 c3)))
law_ExtChoice_Assoc_RtoL c = (False, c)
```

“prefix-⊓-dist” $a \rightarrow (P_1 \sqcap P_2) = (a \rightarrow P_1) \sqcap (a \rightarrow P_2)$

```
law_Prefix_IntChoice_Dist_LtoR (a :-> (IntChoice c1 c2))
    = (True, IntChoice (a :-> c1) (a :-> c2))
law_Prefix_IntChoice_Dist_LtoR c = (False,c)
law_Prefix_IntChoice_Dist_RtoL c@(IntChoice (e@(a,ds)::-> c1) ((a',ds')::->c2))
    | a == a' && ds == ds' = (True, e ::-> (IntChoice c1 c2))
   | otherwise = (False, c)
```

“prefix-Dist” $c \rightarrow (\bigsqcap_{x:A} \bullet P) = \bigsqcap_{x:A} \bullet c \rightarrow P$

```
law_Prefix_Dist_LtoR (e :-> (INT x a circ))
    = (True, INT x a (e :-> circ))
law_Prefix_Dist_LtoR c = (False, c)
law_Prefix_Dist_RtoL (INT x a (e :-> circ))
    = (True, e :-> (INT x a circ))
law_Prefix_Dist_RtoL c = (False, c)
```

“⊓-□-dist-l” $P_1 \sqcap (P_2 \Box P_3) = (P_1 \sqcap P_2) \Box (P_1 \sqcap P_3)$

```
law_IntChoice_ExtChoice_Dist_LtoR (IntChoice c1 (ExtChoice c2 c3))
        = (True, (ExtChoice (IntChoice c1 c2) (IntChoice c1 c3)))
law_IntChoice_ExtChoice_Dist_LtoR c = (False, c)
law_IntChoice_ExtChoice_Dist_RtoL c@(ExtChoice (IntChoice c1 c2) (IntChoice c1' c3))
        | c1 == c1' = (True, IntChoice c1 (ExtChoice c2 c3))
        | otherwise = (False, c)
```

“□-⊓-dist-l”

```
law_ExtChoice_IntChoice_Dist_LtoR (ExtChoice c1 (IntChoice c2 c3))
        = (True, (IntChoice (ExtChoice c1 c2) (ExtChoice c1 c3)))
law_ExtChoice_IntChoice_Dist_LtoR c = (False, c)
law_ExtChoice_IntChoice_Dist_RtoL c@(IntChoice (ExtChoice c1 c2) (ExtChoice c1' c3))
        | c1 == c1' = (True, ExtChoice c1 (IntChoice c2 c3))
        | otherwise = (False, c)
```

“□-Dist”

```
law_ExtChoice_Dist_LtoR (ExtChoice c1 (INT x a c2))
        = (True, INT x a (ExtChoice c1 c2))
law_ExtChoice_Dist_LtoR c = (False, c)
law_ExtChoice_Dist_RtoL (INT x a (ExtChoice c1 c2))
        = (True, ExtChoice c1 (INT x a c2))
law_ExtChoice_Dist_RtoL c = (False, c)
```

"□-unit-l" $STOP \,\Box\, P = P$

```
law_ExtChoice_Unit (ExtChoice Stop c) = (True,c)
law_ExtChoice_Unit c = (False, c)
```

"◁▷-idem" $P \lhd b \rhd P = P$

```
law_Cond_Idem_L circ@(Cond b c c')
    | c == c' = (True, c)
    | otherwise = (False, circ)
law_Cond_Idem_R b circ = (True, Cond b circ circ)
```

"◁▷-dist-l" $(P \sqcap Q) \lhd b \rhd R = (P \lhd b \rhd R) \sqcap (Q \lhd b \rhd R)$

```
law_Cond_Dist_lL (Cond b (IntChoice c1 c2) c3)
    = (True, IntChoice (Cond b c1 c2) (Cond b c2 c3))
law_Cond_Dist_lL c = (False, c)
law_Cond_Dist_lR circ@(IntChoice (Cond b c1 c3) (Cond b' c2 c3'))
    | b == b' && c3 == c3' = (True, Cond b (IntChoice c1 c2) c3)
    | otherwise = (False, circ)
```

"◁▷-dist-r" $R \lhd b \rhd (P \sqcap Q) = (R \lhd b \rhd P) \sqcap (R \lhd b \rhd Q)$

```
law_Cond_Dist_rL (Cond b c3 (IntChoice c1 c2))
    = (True, IntChoice (Cond b c3 c1) (Cond b c3 c2))
law_Cond_Dist_rL c = (False, c)
law_Cond_Dist_rR circ@(IntChoice (Cond b c3 c1) (Cond b' c3' c2))
    | b == b' && c3 == c3' = (True, Cond b c3 (IntChoice c1 c2))
    | otherwise = (False, circ)
```

"*true*-id" $P \lhd true \rhd Q = P$

```
law_True_IdL (Cond (B True) c1 c2) = (True, c1)
law_True_IdL c = (False, c)
law_True_IdR c1 c2 = (True, Cond true c1 c2)
```

"*false*-id" $P \lhd false \rhd Q = Q$

```
law_False_IdL (Cond (B False) c1 c2) = (True, c2)
law_False_IdL c = (False, c)
law_False_IdR c1 c2 = (True, Cond false c1 c2)
```

"$\lhd\rhd$-$\Box$-dist" $P \Box (Q \lhd b \rhd R) = (P \Box Q) \lhd b \rhd (P \Box R)$

```
law_Cond_ExtChoice_DistL (ExtChoice c1 (Cond b c2 c3))
    = (True, Cond b (ExtChoice c1 c2) (ExtChoice c1 c3))
law_Cond_ExtChoice_DistL c = (False, c)
law_Cond_ExtChoice_DistR circ@(Cond b (ExtChoice c1 c2) (ExtChoice c1' c3))
    | c1 == c1' = (True, ExtChoice c1 (Cond b c2 c3))
    | otherwise = (False, circ)
```

"$\|$-dist" $P_X\|_Y (Q \sqcap R) = (P_X\|_Y Q) \sqcap (P_X\|_Y R)$

```
law_APar_Dist_LtoR (APar x y c1 (IntChoice c2 c3))
        = (True, IntChoice (APar x y c1 c2) (APar x y c1 c3))
law_APar_Dist_LtoR c = (False, c)
law_APar_Dist_RtoL circ@(IntChoice (APar x y c1 c2) (APar x' y' c1' c3))
    | x == x' && y == y' && c1 == c1' = (True, APar x y c1 (IntChoice c2 c3))
    | otherwise = (False, circ)
```

"$\|$-sym" $P_X\|_Y Q = Q_Y\|_X P$

```
law_APar_Symm (APar x y c1 c2) = (True, (APar x y c2 c1))
law_APar_Symm c = (False, c)
```

"$\|$-assoc" $(P_X\|_Y Q)_{X\cup Y}\|_Z R = P_X\|_{Y\cup Z} (Q_Y\|_Z R)$

```
law_APar_AssocL x y circ@(APar xuniony z (APar x' y' c1 c2) c3)
        | x == x' && y == y' = (True, APar x yunionz c1 (APar y z c2 c3))
        | otherwise = (False, circ)
        where
                xuniony = x `union` y
                yunionz = y `union` z

law_APar_AssocR y z circ@(APar x yunionz c1 (APar y' z' c2 c3))
        | z == z' && y == y' = (True, APar xuniony z (APar x y c1 c2) c3)
        | otherwise = (False, circ)
        where
                xuniony = x `union` y
                yunionz = y `union` z
```

"$\|$-termination" $SKIP_X\|_Y SKIP = SKIP$

```
law_APar_TerminateL (APar x y Skip Skip) = (True, Skip)
law_APar_TerminateL c = (False, c)
```

"$|||$-sym" $P \mathbin{|||} Q = Q \mathbin{|||} P$

```
law_Ilv_Symm (Ilv c1 c2) = (True, Ilv c2 c1)
law_Ilv_Symm c = (False, c)
```

"$|||$-assoc" $(P \mathbin{|||} Q) \mathbin{|||} R = P \mathbin{|||} (Q \mathbin{|||} R)$

```
law_Ilv_AssocL (Ilv (Ilv c1 c2) c3) = (True, Ilv c1 (Ilv c2 c3))
law_Ilv_AssocL c = (False, c)

law_Ilv_AssocR (Ilv c1 (Ilv c2 c3)) = (True, Ilv (Ilv c1 c2) c3)
law_Ilv_AssocR c = (False, c)
```

"$|||$-dist" $P \mathbin{|||} (Q \sqcap R) = (P \mathbin{|||} Q) \sqcap (P \mathbin{|||} R)$

```
law_Ilv_DistL (Ilv c1 (IntChoice c2 c3))
    = (True, IntChoice (Ilv c1 c2) (Ilv c1 c3))
law_Ilv_DistL c = (False, c)

law_Ilv_DistR circ@(IntChoice (Ilv c1 c2) (Ilv c1' c3))
    | c1 == c1' = (True, Ilv c1 (IntChoice c2 c3))
    | otherwise = (False, circ)
```

"$|||$-unit" $SKIP \mathbin{|||} P = P$

```
law_IlvUnitL (Ilv Skip circ) = (True, circ)
law_IlvUnitL circ = (False, circ)
law_IlvUnitR circ = (True, Ilv Skip circ)
```

"$\|_I$-sym" $P \mathbin{|[I]|} Q = Q \mathbin{|[I]|} P$

```
law_IPar_Symm (IPar x c1 c2) = (True, (IPar x c2 c1))
law_IPar_Symm c = (False, c)
```

"$\|_I$-dist" $P \mathbin{|[X]|} (Q \sqcap R) = (P \mathbin{|[X]|} Q) \sqcap (P \mathbin{|[X]|} R)$

```
law_IPar_Dist_LtoR (IPar x c1 (IntChoice c2 c3))
        = (True, IntChoice (IPar x c1 c2) (IPar x c1 c3))
law_IPar_Dist_LtoR c = (False, c)

law_IPar_Dist_RtoL circ@(IntChoice (IPar x c1 c2) (IPar x' c1' c3))
    | c1 == c1' && x == x' = (True, IPar x c1 (IntChoice c2 c3))
    | otherwise = (False, circ)
```

“$\|_I$-assoc” $P \llbracket X \rrbracket (Q \llbracket X \rrbracket R) = (P \llbracket X \rrbracket Q) \llbracket X \rrbracket R$

```
law_IPar_AssocL (IPar xi c1 (IPar xf c2 c3))
        | xi==xf = (True, (IPar xi (IPar xi c1 c2) c3))
law_IPar_AssocL c = (False, c)

law_IPar_AssocR circ@(IPar x (IPar x' c1 c2) c3)
    | x == x' = (True, IPar x c1 (IPar x c2 c3))
    | otherwise = (False, circ)
```

“$\|$-termination” $SKIP \llbracket X \rrbracket SKIP = SKIP$

```
law_IPar_TerminateL (IPar x Skip Skip) = (True, Skip)
law_IPar_TerminateL c = (False, c)
```

“$\setminus$-dist” $(P \sqcap Q) \setminus X = (P \setminus X) \sqcap (Q \setminus X)$

```
law_Hide_DistL (Hide (IntChoice c1 c2) x)
        = (True, (IntChoice (Hide c1 x) (Hide c2 x)))
law_Hide_DistL c = (False, c)

law_Hide_DistR circ@(IntChoice (Hide c1 x) (Hide c2 x'))
    | x == x' = (True, Hide (IntChoice c1 c2) x)
    | otherwise = (False, circ)
```

“$\setminus$-sym” $(P \setminus Y) \setminus X = (P \setminus X) \setminus Y$

```
law_Hide_Symm (Hide (Hide c x) y) = (True, Hide (Hide c y) x)
law_Hide_Symm c = (False, c)
```

“$\setminus$-combine” $(P \setminus Y) \setminus X = P \setminus (X \cup Y)$

```
law_Hide_CombineL (Hide (Hide c x) y)
= (True, (Hide c xuniony))
    where
        xuniony = x `union` y
law_Hide_CombineL c = (False, c)

law_Hide_CombineR x y (Hide c xuniony)
= (True, Hide (Hide c x) y)
    where
        xuniony = x `union` y
law_Hide_CombineR x y c = (False, c)
```

"$SKIP$-\-id" $SKIP \setminus X = SKIP$

```
law_Hide_SkipIdL (Hide Skip x) = (True, Skip)
law_Hide_SkipIdL c = (False, c)
law_Hide_SkipIdR x Skip = (True, Hide Skip x)
```

"\-null" $P \setminus \{\} = P$

```
law_Hide_NullL (Hide c setnull) = (True, c)
law_Hide_NullL c = (False, c)
law_Hide_NullR c = (True, Hide c [])
```

"\-||-dist" $(P\,{}_X\|_Y Q) \setminus Z = (P \setminus Z \cap X)_X\|_Y(Q \setminus Z \cap Y)$

```
law_Hide_APar_DistL (Hide (APar x y c1 c2) z)
    = (True, APar x y (Hide c1 zintsctx) (Hide c2 zintscty))
      where
         zintsctx = z `intersect` x
         zintscty = z `intersect` y

law_Hide_APar_DistL c = (False, c)

law_Hide_APar_DistR z x' y' z' circ@(APar x y (Hide c1 zintsctx')
                                              (Hide c2 z'intscty'))
    | x==x' && y==y' && z==z' = (True, Hide (APar x y c1 c2) z)
    | otherwise = (False, circ)
      where
         zintsctx' = z `intersect` x'
         z'intscty' = z' `intersect` y'
```

"\-||-dist" $(P \,[\![X]\!]\, Q) \setminus Z = (P \setminus Z) \,[\![X]\!]\, (Q \setminus Z), \quad X \cap Z = \{\}$

```
law_Hide_IPar_DistL circ@(Hide (IPar x c1 c2) z)
     | xintsctz == [] = IPar x (Hide c1 z) (Hide c2 z)
     | otherwise = circ
     where
        xintsctz = x `intersect` z

law_Hide_IPar_DistR circ@(IPar x (Hide c1 z) (Hide c2 z'))
    | xintsctz == [] && z == z' = (True, Hide (IPar x c1 c2) z)
    | otherwise = (False, circ)
      where
         xintsctz = x `intersect` z
```

"; -dist-l" $(P \sqcap Q) \fatsemi R = (P \fatsemi R) \sqcap (Q \fatsemi R)$

```
law_Seq_Ic_DistlL ((IntChoice c1 c2) ::: c3)
    = (True, IntChoice (c1 ::: c3) (c2 ::: c3))
law_Seq_Ic_DistlL c = (False, c)

law_Seq_Ic_DistlR circ@(IntChoice (c1 ::: c3) (c2 :::c3'))
    | c3 == c3' = (True, (IntChoice c1 c2) ::: c3)
    | otherwise = (False, circ)
```

“; -dist-r” $P \mathbin{;} (Q \sqcap R) = (P \mathbin{;} Q) \sqcap (P \mathbin{;} R)$

```
law_Seq_Ic_DistrL (c1 ::: (IntChoice c2 c3))
    = (True, IntChoice (c1 ::: c2) (c1 ::: c3))
law_Seq_Ic_DistrL c = (False, c)

law_Seq_Ic_DistrR  circ@(IntChoice (c1 ::: c2) (c1' ::: c3))
    | c1 == c1' = (True, c1 ::: (IntChoice c2 c3))
    | otherwise = (False, circ)
```

C.5.2 Step Laws

“*STOP*-step” $a : \{\} \rightarrow P = STOP$

```
law_Stop_StepL ((x,setnull) ::-> c) = (True, Stop)
law_Stop_StepL c = (False, c)
```

```
foldEc :: [Circus] -> Circus
foldEc [] = Stop
foldEc [c] = c
foldEc (c:cs) = ExtChoice c (foldEc cs)
```

“□-step”

$$\begin{array}{ll} & x : A \rightarrow P(x) \,\square\, y : B \rightarrow Q(y) \qquad \text{“□-step”} \\ = & x : (A \setminus B) \rightarrow P(x) \\ \square & z : (A \cap B) \rightarrow (P(z) \sqcap Q(z)) \\ \square & y : (B \setminus A) \rightarrow Q(y) \end{array}$$

```
law_ExtChoice_StepL circ@(ExtChoice ((x,xs) ::-> c1) ((y,ys) ::-> c2))
        = (True, mkExtChoice p1 (mkExtChoice p2 p3))
        where
                p1 = (x,adiffb) --> c1
                p2 = (z,aintsctb) --> (mkIntChoice p4 p5)
                p3 = (y,bdiffa) --> c2
                adiffb = xs \\ ys
                aintsctb = xs `intersect` ys
                bdiffa = ys \\ xs
```

```
                 p4 = (csubstitute x z c1)
                 p5 = (csubstitute y z c2)
                 z = freshName $ circNames circ
law_ExtChoiceStepL c = (False, c)

--test bench ExtChoiceStep
tbec x xs y ys = law_ExtChoice_StepL (ExtChoice ((x,xs)::->Skip) ((y,ys)::->Skip))
```

“$\|$-step”

$$\begin{array}{ll} & x : C \rightarrow P(x)\, {}_A\|_B\, y : D \rightarrow Q(y) \\ = & x : ((C \cap A) \setminus B) \rightarrow (P(x)\, {}_A\|_B\, y : D \rightarrow Q(y)) \\ \Box & z : (C \cap A \cap D \cap B) \rightarrow (P(z)\, {}_A\|_B\, Q(z)) \\ \Box & y : ((D \cap B) \setminus A) \rightarrow (x : C \rightarrow P(x)\, {}_A\|_B\, Q(y)) \end{array} \quad \text{“}\|\text{-step”}$$

```
law_APar_StepL circ@(APar a b ((x,xs)::->c1) ((y,ys)::->c2))
        = (True, mkExtChoice p1 (mkExtChoice p2 p3))
        where
           p1 = ((x,comb1) --> (APar a b c1 ((y,ys)::->c2)))
           p2 = ((z,comb2) --> (APar a b c3 c4))
           p3 = ((y,comb3) --> (APar a b c5 c2))
           comb1 = ((xs `intersect` a) \\ b)
           comb2 = ((xs `intersect` (a `intersect` (ys `intersect` b))))
           comb3 = ((ys `intersect` b) \\ a)
           c3 = (csubstitute x z c1)
           c4 = (csubstitute y z c2)
           c5 = ((x,xs)-->c1)
           z = freshName $ circNames circ
law_APar_StepL c = (False, c)

-- Test Bench AParStep
tbapar a b x xs y ys = law_APar_StepL (APar a b ((x,xs)::-> Skip) ((y,ys)::->Skip))
```

“$\|_A$-step”

$$\begin{array}{ll} & x : C \rightarrow P(x) \,\|_A\, y : D \rightarrow Q(y) \\ = & x : (C \setminus (D \cup A)) \rightarrow (P(x) \,\|_A\, y : D \rightarrow Q(y)) \\ \Box & z : (C \cap A \cap D) \rightarrow (P(z) \,\|_A\, Q(z)) \\ \Box & y : (D \setminus (C \cup A)) \rightarrow (x : C \rightarrow P(x) \,\|_A\, Q(y)) \\ \Box & z : ((C \cap D) \setminus A) \rightarrow (\,(P(x) \,\|_A\, y : D \rightarrow Q(y)) \\ & \qquad \sqcap \\ & \qquad (x : C \rightarrow P(x) \,\|_A\, Q(y))\,) \end{array} \quad \text{“}\|_A\text{-step”}$$

```
law_IPar_StepL circ@(IPar a ((x,xs)::->c1) ((y,ys)::->c2))
        = (True, mkExtChoice p1 (mkExtChoice p2 (mkExtChoice p3 p4)))
        where
           p1 = ((x,comb1) --> (IPar a c1 ((y,ys)::->c2)))
```

```
            p2 = ((z,comb2) --> (IPar a c3 c4))
            p3 = ((y,comb3) --> (IPar a ((x,xs)::->c1) c2))
            p4 = ((z,comb4) --> (mkIntChoice (IPar a c1 ((y,ys)::->c2))
                                          (IPar a ((x,xs)::->c1) c2)))
            comb1 = (xs \\ (ys `union` a))
            comb2 = (xs `intersect` (a `intersect` ys))
            comb3 = (ys \\ (xs `union` a))
            comb4 = ((xs `intersect` ys) \\ a)
            c3 = (csubstitute x z c1)
            c4 = (csubstitute y z c2)
            z = freshName $ circNames circ
law_IPar_StepL c = (False, c)

--Test Bench IParStep
tbipar a x xs y ys = law_IPar_StepL (IPar a ((x,xs)::->Skip) ((y,ys)::->Skip))
```

"$|||$-step"

$$\begin{array}{ll} x : C \to P(x) \,|||\, y : D \to Q(y) & \text{"}|||\text{-step"} \\ \quad = \quad x : C \to (P(x) \,|||\, y : D \to Q(y)) & \\ \quad \sqcap \quad y : D \to (x : C \to P(x) \,|||\, Q(y)) & \end{array}$$

```
law_ILeave_StepL (Ilv ((x,xs)::->c1) ((y,ys)::->c2))
        = (True, mkIntChoice p1 p2)
        where
                p1 = ((x,xs)-->(Ilv c1 ((y,ys)-->c2)))
                p2 = ((y,ys)-->(Ilv ((x,xs)-->c1) c2))
law_ILeave_StepL c = (False, c)

--Test Bench ILeave
tbilv x xs y ys = law_ILeave_StepL (Ilv ((x,xs)::->Skip) ((y,ys)::->Skip))
```

"$\backslash$-step"

$$(a \to P) \setminus H = \begin{cases} a \to (P \setminus H), & \text{if } a \notin H \\ P \setminus H, & \text{if } a \in H \end{cases}$$

We generalise this to:

$$(a : A \to P) \setminus H = (a : (A \setminus H) \to (P \setminus H)) \sqcap \textstyle\bigsqcap_{a:A\cap H} \bullet (P \setminus H)$$

```
law_Hide_StepL (Hide ((x,xs) ::-> c) hdn) = (True, mkIntChoice p1 p2)
        where
                p1 = ((x, euh) --> puh)
                p2 = mkINT x aintrscth (Hide c hdn)
                puh = (Hide c hdn)
                euh = xs \\ hdn
                aintrscth =  xs `intersect` hdn
```

```
law_Hide_StepL c = (False, c)

--Test Bench HideStep
tbhide x xs hdn = law_Hide_StepL (Hide ((x,xs)::-> Skip) hdn)
```

“; -step”

$$\begin{array}{ll} (x : C \rightarrow P(x));\ Q & \text{“; -step”} \\ \quad =\quad y : C \rightarrow (P(y);\ Q), \quad y \text{ fresh} & \end{array}$$

```
law_Seq_StepL circ@(((x,xs)::->c1) ::: c2) =
   (True, (x,(map (esubstitute y x) xs))::->(c1 ::: c2))
   where
      y = freshName $ circNames circ
law_Seq_StepL c = (False, c)
--Test Bench SeqStep
tbseq x xs = law_Seq_StepL (((x,xs)::->Skip) ::: Skip)
```

C.6 Simple Circus Translation

```
module SimpleCircusTranslate where
import Data.List

import Maybe
import Utilities
import SimpleCircus
import CircusNameMgmt
import StdCircusNames
import SimpleCircusLaws
```

C.6.1 Normal Form

```
nf ((a:->Skip)::: (ExtChoice c1 c2))  = (a :-> Skip) ::: ExtChoice (nf c1) (nf c2)
nf ((ExtChoice c1 c2) ::: c3) = ExtChoice (nf $ nfseq (nf c1) (nf c3))
(nf $ nfseq (nf c2) (nf c3))
nf (a :-> (IntChoice c1 c2))  = IntChoice (a :-> nf c1) (a :-> nf c2)
nf (a :-> c) = a :-> nf c
nf (c1 ::: c2) = nf $ nfseq (nf c1) (nf c2)
nf (ExtChoice c1 c2) = ExtChoice (nf c1) (nf c2)
nf (IntChoice c1 c2) = IntChoice (nf c1) (nf c2)
nf c = c

nfseq Stop _ = Stop
nfseq Skip c = c
nfseq c Skip = c
nfseq (a :-> nfc1) nfc2 = a :-> (nfseq nfc1 nfc2)
```

C.6.2 Acquiring Head Normal Form

Following are already in hnf.

```
hnf Stop = Stop
hnf Skip = Skip
hnf Div = Div
hnf circ@((x,xs)::->c) = circ
```

For internal choice required to be in hnf,
$hnf(Q \sqcap R) \mathrel{\widehat{=}} hnf(Q) \sqcap hnf(R)$

```
hnf (IntChoice c1 c2) = mkIntChoice (hnf c1) (hnf c2)
```

“□-step”

$$
\begin{array}{ll}
& x : A \to P(x) \,\Box\, y : B \to Q(y) \qquad \text{“□-step”} \\
= & x : (A \setminus B) \to P(x) \\
\Box & z : (A \cap B) \to (P(z) \sqcap Q(z)) \\
\Box & y : (B \setminus A) \to Q(y)
\end{array}
$$

```
hnf (ExtChoice c1 c2) = (snd $ law_ExtChoice_StepL
         (mkExtChoice (snd $ law_ExtChoice_IntChoice_Dist_LtoR (hnf c1))
 (snd $ law_ExtChoice_IntChoice_Dist_LtoR (hnf c2))))
```

“$\|_A$-step”

$$
\begin{array}{ll}
& x : C \to P(x) \,\|_A\, y : D \to Q(y) \qquad \text{“}\|_A\text{-step”} \\
= & x : (C \setminus (D \cup A)) \to (P(x) \,\|_A\, y : D \to Q(y)) \\
\Box & z : (C \cap A \cap D) \to (P(z) \,\|_A\, Q(z)) \\
\Box & y : (D \setminus (C \cup A)) \to (x : C \to P(x) \,\|_A\, Q(y)) \\
\Box & z : ((C \cap D) \setminus A) \to (\, (P(x) \,\|_A\, y : D \to Q(y)) \\
& \qquad \sqcap \\
& \qquad (x : C \to P(x) \,\|_A\, Q(y)) \,)
\end{array}
$$

```
hnf (IPar x c1 c2) = (snd $ law_IPar_StepL (IPar x (snd $ law_IPar_Dist_LtoR (hnf c1))
         (snd $ law_IPar_Dist_LtoR (hnf c2))))
```

“$\|$-step”

$$
\begin{array}{ll}
& x : C \to P(x) \,{}_A\|_B\, y : D \to Q(y) \qquad \text{“}\|\text{-step”} \\
= & x : ((C \cap A) \setminus B) \to (P(x) \,{}_A\|_B\, y : D \to Q(y)) \\
\Box & z : (C \cap A \cap D \cap B) \to (P(z) \,{}_A\|_B\, Q(z)) \\
\Box & y : ((D \cap B) \setminus A) \to (x : C \to P(x) \,{}_A\|_B\, Q(y))
\end{array}
$$

```
hnf (APar x y c1 c2) = (snd $ law_APar_StepL (APar x y (snd $ law_APar_Dist_LtoR (hnf c1))
         (snd $ law_APar_Dist_LtoR (hnf c2))))
hnf c = c
```

C.6.3 Translation

Here, we briefly explain the purpose and implementation details of each defined function as it appears in the translator code.

`isCSP` is a function which takes a Circus action and returns a boolean indicating if the particular *Circus* action is a *CSP* one or not.

```
isCSP :: Circus -> Bool  -- to be implemented later
isCSP _ = False
```

`action2csp` is a high level function which takes a complete circus program and a defined action in it and returns its equivalent in the *CSP* world. The implementation is achieved by making use of `mkCGraph` appearing later in the implementation. After getting the graph of a particular *Circus* program using `mkCGraph`, the `translateCirc` function is used to do the actual translation based on the action name received and its corresponding variables and calls.

```
action2csp :: CircusProgram -> String -> CircusProgram
action2csp prog aname
 = let cgrf = mkCGraph prog
   in case alookup cgrf aname of
        Nothing  ->  error ("No action '"++aname++"' found")
        Just (vars,calls)  -> translateCirc prog cgrf aname vars calls
```

The difference between high level functions `circus2csp` and `action2csp` is that the function `circus2csp` maps a complete Circus program to its translated version while the latter works on an individual action inside the Circus program.

```
circus2csp :: CircusProgram -> CircusProgram
circus2csp prog
 = let cgrf = mkCGraph prog
   in prog
```

Function `getCircVarsCalls` takes a complete Circus program and gets the information involved i.e. the definition name, the array of maintaining variables used in the definition and the array having the information of the calls of particular actions. This is achieved by using functions `actionVars` and `actionCalls`.

```
getCircVarsCalls :: CircusProgram
            -> [ ( String  -- definition name
                 , ( [String] -- variables used in definition (sorted)
                   , [String] -- actions called
                   )
                 )
               ]
getCircVarsCalls defs
 = alnorm $ gCVC defs
 where
  gCVC [] = []
  gCVC ((aname,(aparam,abody)):rest)
   = (aname,(avars,acalls)):gCVC rest
   where
    avars = lnorm $ actionVars abody
    acalls = lnorm $ actionCalls abody
```

The function `actionVars` takes a particular Circus action and generates an array for gathering names of the

variables used in a particular action.

```
actionVars :: Circus -> [String]

actionVars (x := e) = [x] ++ exprVars e
actionVars (Call act _) = []
actionVars ((x,_) ::-> circ) = x : actionVars circ

actionVars (INT x _ circ) = x : actionVars circ
actionVars (SEQ x _ circ) = x : actionVars circ
actionVars (EXT x _ circ) = x : actionVars circ
actionVars (ILV x _ circ) = x : actionVars circ
actionVars (IPAR _ x _ circ) = x : actionVars circ
actionVars (APAR x _ _ circ) = x : actionVars circ

actionVars (a :-> circ) = actionVars circ
actionVars (c1 ::: c2) = actionVars c1 ++ actionVars c2
actionVars (IntChoice c1 c2) = actionVars c1 ++ actionVars c2
actionVars (ExtChoice c1 c2) = actionVars c1 ++ actionVars c2
actionVars (Hide circ _) = actionVars circ
actionVars (IPar _ c1 c2) = actionVars c1 ++ actionVars c2
actionVars (APar _ _ c1 c2) = actionVars c1 ++ actionVars c2
actionVars (Ilv c1 c2) = actionVars c1 ++ actionVars c2
actionVars (Cond e c1 c2) = exprVars e ++ actionVars c1 ++ actionVars c2
actionVars (Guard e circ) = exprVars e ++ actionVars circ
actionVars (Mu _ circ) = actionVars circ
actionVars _ = []
```

The function `actionCalls` takes a particular Circus action and generates an array for gathering names of the calls to particular actions.

```
actionCalls :: Circus -> [String]

actionCalls (Call a _) = [a]
actionCalls (n := e) = []

actionCalls (c1 ::: c2) = actionCalls c1 ++ actionCalls c2
actionCalls (a :-> circ) = actionCalls circ
actionCalls ((x,xs) ::-> circ) = actionCalls circ
actionCalls (IntChoice c1 c2) = actionCalls c1 ++ actionCalls c2
actionCalls (ExtChoice c1 c2) = actionCalls c1 ++ actionCalls c2
actionCalls (Hide circ a) = actionCalls circ
actionCalls (IPar a c1 c2) = actionCalls c1 ++ actionCalls c2
actionCalls (APar a b c1 c2) = actionCalls c1 ++ actionCalls c2
actionCalls (Ilv c1 c2) = actionCalls c1 ++ actionCalls c2
actionCalls (Cond e c1 c2) = actionCalls c1 ++ actionCalls c2
actionCalls (Guard e circ) = actionCalls circ
actionCalls (INT x a circ) = actionCalls circ
actionCalls (SEQ x a circ) = actionCalls circ
actionCalls (EXT x a circ) = actionCalls circ
actionCalls (ILV x a circ) = actionCalls circ
```

```
actionCalls (IPAR _ x _ circ) = actionCalls circ
actionCalls (APAR x _ _ circ) = actionCalls circ
actionCalls (Mu x circ) = actionCalls circ \\ [x]
actionCalls _ = []
```

To gather information on the variables used in a particular expression, the function `exprVars` is defined.

```
exprVars (Var v) = [v]
exprVars (Agg _ _ es) = concat $ map exprVars es
exprVars (App _ es) = concat $ map exprVars es
exprVars (Bin _ e1 e2) = exprVars e1 ++ exprVars e2
exprVars _ = []
```

The important function here is `detCircDeps` which manages the record of the dependencies for the final translated version of the Circus program. It determines the dependencies by analysing the calls to particular actions by using `getCalls` function.

```
detCircDeps :: [(String,([String],[String]))] -> [(String,([String],[String]))]
detCircDeps deps
 = dCP deps [] False deps
 where

   dCP deps0 deps’ chgd []
     | chgd  =  dCP deps’ [] False deps’
     | otherwise  =  deps’ -- should equal deps0 !

   dCP deps0 deps’ chgd  (dep@(name,(vars,calls)):rest)
     | calls’ == calls  =  dCP deps0 (dep:deps’) chgd rest
     | otherwise  =  dCP deps0 ((name,(vars,calls’)):deps’) True rest
     where calls’  = getCalls deps0 calls calls

getCalls :: [(String,([String],[String]))] -> [String] -> [String] -> [String]
getCalls deps0 calls’ [] = lnorm calls’
getCalls deps0 calls’ (call:calls)
 = case alookup deps0 call of
     Nothing  ->  error ("Action ’"++call++"’ is undefined")
     Just (_,subcalls) ->  getCalls deps0 (subcalls++calls’) calls
```

As mentioned earlier, `mkCGraph` and `translateCirc` are used by the high level functions of translator called `action2csp` and `circus2csp`. After getting the graph of a particular Circus program through `mkCGraph`, the `translateCirc` function is used to do the actual translation based on the action name received and its corresponding variables and calls.

```
mkCGraph :: CircusProgram -> [(String,([String],[String]))]
mkCGraph = detCircDeps . getCircVarsCalls
```

```
translateCirc :: CircusProgram -> [(String,([String],[String]))]
                               -> String -> [String] -> [String] -> CircusProgram
translateCirc prog cgrf aname vars calls
 = let usedActionNames = lnorm (aname:calls)
       isUsed (nm,_) = nm `elem` usedActionNames
       rprog = filter  isUsed prog

       newplist = lnorm (vars ++ getParams cgrf calls)
       plistvars = map Var newplist
       addpars (nm,(pars,body))
        = (nm,(pars++newplist,addParams nm plistvars body))

       pprog = map addpars rprog

   in pprog
```

The purpose of the `addParams` function is to attach the list of parameters to a *Circus* action. This is required because in the *CSP* world the variables of the *Circus* world turn into parameters and a particular action is invoked using parametric calls. While dealing with the assignment commands of the *Circus* world, the expression assigned to a particular variable is substituted in the parameter list.

```
addParams :: String -> [Expr] -> Circus -> Circus
addParams nm plist (Call cnm pars) = (Call cnm (pars++plist))
addParams nm plist Skip = (Call (nm++"_CONT") plist)
addParams nm plist (c1 ::: c2) = (addParams nm plist c1 ::: addParams nm plist c2)
addParams nm plist (a :-> circ) = (a :-> (addParams nm plist circ))
addParams nm plist ((x,xs) ::-> circ) = ((x,xs) ::-> addParams nm plist circ)
addParams nm plist (IntChoice c1 c2) = (IntChoice (addParams nm plist c1)
                                                  (addParams nm plist c2))
addParams nm plist (ExtChoice c1 c2) = (ExtChoice (addParams nm plist c1)
                                                  (addParams nm plist c2))
addParams nm plist (Hide circ a) = (Hide (addParams nm plist circ) a)
addParams nm plist (IPar a c1 c2) = (IPar a (addParams nm plist c1)
                                            (addParams nm plist c2))
addParams nm plist (APar a b c1 c2) = (APar a b (addParams nm plist c1)
                                                (addParams nm plist c2))
addParams nm plist (Ilv c1 c2) = (Ilv (addParams nm plist c1)
                                      (addParams nm plist c2))
addParams nm plist (Cond e c1 c2) = (Cond e (addParams nm plist c1)
                                            (addParams nm plist c2))
addParams nm plist (Guard e circ) = (Guard e (addParams nm plist circ))
addParams nm plist (INT x a circ) = (INT x a (addParams nm plist circ))
addParams nm plist (SEQ x a circ) = (SEQ x a (addParams nm plist circ))
addParams nm plist (EXT x a circ) = (EXT x a (addParams nm plist circ))
addParams nm plist (ILV x a circ) = (ILV x a (addParams nm plist circ))
addParams nm plist (IPAR a x b circ) = (IPAR a x b (addParams nm plist circ))
addParams nm plist (APAR x a b circ) = (APAR x a b (addParams nm plist circ))
addParams nm plist body = body
```

```
-- extract part of prog of interest - all definitions aname:calls (rprog)
-- add newplist to extend plist of every call (pprog)
-- return pprog

getParams :: [(String,([String],[String]))] -> [String] -> [String]
getParams cgrf calls = concat $ map (fst . fromJust . alookup cgrf)  calls
```

```
getCallChains :: CircusProgram -> [ [ [String] ] ]

getCallChains defs
   = gCC defs
 where
  gCC [] = []
  gCC ((aname,(aparam,abody)):rest)
   | aname == "MAIN" = (normTreeSt):gCC rest
   | otherwise = gCC rest
   where
     normTreeSt = normTree abody

normTree :: Circus -> [[String]]

normTree (ExtChoice c1 c2) = [[ecName]] ++ [actionCallChains c1] ++ [actionCallChains c2]
normTree (IntChoice c1 c2) = [[icName]] ++ [actionCallChains c1] ++ [actionCallChains c2]
normTree (c1 ::: c2) = [[seqName]] ++ [actionCallChains c1 ++ actionCallChains c2]

normTree _ = [[]]

actionCallChains :: Circus -> [String]

actionCallChains (Call a _) = [a]
actionCallChains  (c1 ::: c2) = actionCallChains c1 ++ actionCallChains c2
actionCallChains _ = []
```

The function `getCallTree` is defined here to create the call tree after analysing the `MAIN` action. It is assumed here that the *Circus* main is constituted only of calls to actions, internal choice and external choice.

```
class Functor1 f where
  fmap1 :: (a -> b) -> f a -> f b

data Tree a = Empty | Node (Tree a) String (Tree a) | Leaf a
   deriving (Eq, Ord, Show)

instance Functor1 Tree where
  fmap1 f (Node l m r) = Node (fmap1 f l) m (fmap1 f r)
  fmap1 f (Leaf x) = Leaf (f x)

getCallTree :: CircusProgram -> [ Tree String ]
getCallTree defs
   = gCT defs
```

```
 where
  gCT [] = []
  gCT ((aname,(aparam,abody)):rest)
   | aname == "MAIN" = (callTreeSt):gCT rest
   | otherwise = gCT rest
   where
     callTreeSt = callTree abody

callTree :: Circus -> Tree String

callTree (Call a _) = Leaf a
callTree (ExtChoice c1 c2)  =  (Node (callTree c1) ecName (callTree c2))
callTree (IntChoice c1 c2)  =  (Node (callTree c1) icName (callTree c2))
callTree (c1 ::: c2) = (Node (callTree c1) seqName (callTree c2))

callTree _ = Empty
```

The function `extractSeq` is defined here to generate fresh names for the Circus actions on the left and right side of the sequential composition. Each leading node from the tree of sequential composition is given a fresh name to make the circus action compositions a sequential one. For example:
$(A[]B);\ (C[]D) => N_1;\ N_2$ where $N_1 = (A[]B)$ and $N_2 = (C[]D)$.

```
getExtractSeq :: CircusProgram -> [ (Circus, [(String, Circus)]) ]
getExtractSeq defs
   = gES defs
 where
  gES [] = []
  gES ((aname,(aparam,abody)):rest)
   | aname == "MAIN" = (extractSeqSt):gES rest
   | otherwise = gES rest
   where
     extractSeqSt = extractSeq abody

extractSeq :: Circus -> (Circus, [(String,Circus)])
extractSeq circ = (circ, newNamedActs circ)

newNamedActs :: Circus -> [(String, Circus)]

newNamedActs (ExtChoice c1 c2) = [(newActNameGen c1, c1)] ++ [(newActNameGen c2, c2)]
newNamedActs (IntChoice c1 c2) = [(newActNameGen c1, c1)] ++ [(newActNameGen c2, c2)]
newNamedActs (c1 ::: c2) = (newNamedActs c1) ++ (newNamedActs c2)

newNamedActs _ = []

newActNameGen :: Circus -> String

newActNameGen circ = (freshNameAct $ circNames circ)
```

C.7 Simple Circus Examples

```
module StdCircusExamples where
import Data.List
import Maybe
import Utilities
import SimpleCircus
import StdCircusNames
import CircusNameMgmt
import SimpleCircusLaws
import StdCircusPrint
```

C.7.1 Examples

```
a = "a"
b = "b"
c = "c"
ex1 = (a :-> Skip) ::: (b :-> Skip) ::: (c :-> Skip)
ex2 = (a :-> Skip) ::: (b :-> Stop) ::: (c :-> Skip)
ex3 = (a :-> Skip) ::: (IntChoice (b:->Skip) (c:->Skip))
ex4 = (a :-> Skip) ::: (ExtChoice (b:->Skip) (c:->Skip))
ex5 = (ExtChoice (a:->Skip) (b:->Skip)) ::: (c:->Skip)
-- For checking step law for external choice
in6 = ExtChoice (("x",["f","g"])::->Skip)
                (("y",["g","i","j","k"])::->Skip)
ex6 = law_ExtChoice_StepL in6
-- For checking step law for alphabetised parallel
ex7 = APar ["e","f"] ["e","g"]
           (("h",["i","j"])::-> Skip)
           (("i",["k","l"])::->Skip)
doex7 = law_APar_StepL ex7
-- For checking step law for interface parallel
ex8 = law_IPar_StepL (IPar ["e","f"] (("e",["f","g"])::->Skip)
      (("e",["g","h","i"])::->Skip))
-- For checking step law for hiding
ex9 = law_Hide_StepL (Hide (("h",["i","j","k"])::->Skip) ["i","j"])
-- Fresh name generation
ex10 = freshName $ circNames (("a0",["b0","c0"])::->Skip)
ex11 = freshName $ circNames (snd $ ex6)
ex12 = freshName $ circNames (snd $ doex7)
ex13 = freshName $ circNames (snd $ ex8)
ex14 = freshName $ circNames (snd $ ex9)
-- example of ppComms
exEvts =  putpp ((Evts ("ch",[Dot (Z 3),Bang (Z 5),Q "v" Nothing])):::->Skip)
```

Examples of Circus Paragraphs

```
add = Bin addName
```

```
sub = Bin subName

cp1 = [  ("A",([],("x" := Z 1) ::: ("a" :-> Skip)))
      , ("B",([],("y" := add (Var "x") (Z 1)) ::: ("b" :-> Skip)))
      , ("C",([],"c" :-> Skip))
      , ("MAIN",([],("z" := Z 0) ::: Call "A" [] ::: Call "B" [] ::: Call "C" []))
      ]
cp2 = [  ("A",([],("x" := Z 1) ::: ("a" :-> Skip)))
      , ("B",([],("y" := add (Var "x") (Z 1)) ::: ("b" :-> Skip)))
      , ("C",([],"c" :-> Skip))
      , ("MAIN",(["P","Q"],("z" := Z 0) ::: Call "A" [] ::: Call "B" [] ::: Call "C" []))
      ]
cp3 = [ ("A",([],("x" := Z 1) ::: ("a" :-> Call "B" [])))
      , ("B",([],("y" := add (Var "x") (Z 1)) ::: ("b" :-> Call "C" [])))
      , ("C",([],"c" :-> Call "A" []))
      ]
cp4 = [ ("A",([],("x" := Z 1) ::: ("a" :-> Call "B" [])))
      , ("B",([],("y" := add (Var "x") (Z 1)) ::: ("b" :-> Call "C" [])))
      , ("C",([],"c" :-> Call "A" []))
      , ("D",([],("a" := add (Var "b") (Z 1)) ::: ("d" :-> Call "D" [])))
      , ("E",([],"e" :-> Call "D" []))
      ]
cp7 = [ ("A",([],("x" := Z 1) ::: ("a" :-> Call "B" [])))
      , ("B",([],("y" := add (Var "x") (Z 1)) ::: ("b" :-> Call "C" [])))
      , ("C",([],"c" :-> Call "A" []))
      , ("D",([],("a" := add (Var "b") (Z 1)) ::: ("d" :-> Call "D" [])))
      , ("E",([],"e" :-> Call "D" []))
      , ("MAIN",([],ExtChoice (Call "A" [] ::: Call "B" [] ::: Call "C" [])
                              (Call "D" [] ::: Call "E" [])))
      ]
cp8 = [ -- creating tree for A ; ( B [] C )
        ("MAIN",([], (Call "A" [] ::: (ExtChoice (Call "B" []) (Call "C" []) ))))
      ]
cp9 = [  -- depicting a complex action composition in MAIN
         -- A;(B;(C;D[]E)[](F;(G[]H))) []  (I;(J[]K))
         ("MAIN",([], ExtChoice
                        (
                          Call "A" [] ::: (Call "B" [] :::
                              (ExtChoice
                                  (Call "C" [] ::: ExtChoice (Call "D" []) (Call "E" []))
                                  (Call "F" [] ::: ExtChoice (Call "G" []) (Call "H" [])))
                                             )
                        )
                        (Call "I" [] ::: (ExtChoice (Call "J" []) (Call "K" []) ))
                )
         )
      ]
```

The lift model:

```
cp5 = [ ("INITLIFT", ([], ("floor" := (Z 1)) ::: ("doorState" := (Val "closed")) ))
      ,
        ("LIFT", ([], foldEc [(Guard (land (lt (Var "floor") (Z 5))
```

```
                                         (eq (Var "doorState") (Val "closed")))
         ("up" :-> ("floor" := ((add (Var "floor") (Z 1)))) ::: Call "LIFT" []))
,
 (Guard (land (gt (Var "floor") (Z 0))
                                        (eq (Var "doorState") (Val "closed")))
         ("down" :-> ("floor" := ((sub (Var "floor") (Z 1)))) ::: Call "LIFT" []))
            ,
              (Guard (eq (Var "doorState") (Val "opened"))
         ("close" :-> ("doorState" := (Val "closed"))) ::: Call "LIFT" [])
,
  (Guard (eq (Var "doorState") (Val "closed"))
         ("open" :-> ("doorState" := (Val "opened"))) ::: Call "LIFT" [])]
  )
)
      ,
        ("MAIN", ([], (Call "INITLIFT" [] ::: Call "LIFT" [])))
      ]
```

C.8 "Standard" Simple Circus

```
module
  StdSimpleCircus
   (module SimpleCircus
   ,module StdCircusNames
   ,module StdCircusPrint
   ,module SimpleCircusLaws
   ,module SimpleCircusTranslate
   ,module StdCircusExamples
   )
where
import Data.List
import Maybe
import Utilities

import SimpleCircus
import StdCircusNames
import StdCircusPrint
import SimpleCircusLaws
import SimpleCircusTranslate
import StdCircusExamples
```

C.9 Implementation of Formalised Steps in Translation Theory

Rename Hidden Events

```
renameHidNames :: Circus -> (Circus,[String])

renameHidNames circ@(x := e) = (circ,[])
renameHidNames circ@(Call act _) = (circ,[])

renameHidNames c@(Hide circ hdn) = (rHN' c hdn)

renameHidNames c@(INT x _ circ) = (c,snd $ renameHidNames circ)
renameHidNames c@(SEQ x _ circ) = (c,snd $ renameHidNames circ)
renameHidNames c@(EXT x _ circ) = (c,snd $ renameHidNames circ)
renameHidNames c@(ILV x _ circ) = (c,snd $ renameHidNames circ)
renameHidNames c@(IPAR _ x _ circ) = (c,snd $ renameHidNames circ)
renameHidNames c@(APAR x _ _ circ) = (c,snd $ renameHidNames circ)

renameHidNames c@(a :-> circ) = (c,snd $ renameHidNames circ)
renameHidNames c@(c1 ::: c2) = ( c,
                                 (snd $ renameHidNames c1) ++
                                 (snd $ renameHidNames c2)
                               )
renameHidNames c@(IntChoice c1 c2) = ( c,
                                 (snd $ renameHidNames c1) ++
                                 (snd $ renameHidNames c2)
                                     )
renameHidNames c@(ExtChoice c1 c2) = ( c,
                                       (snd $ renameHidNames c1) ++
                                       (snd $ renameHidNames c2)
                                     )

renameHidNames c@(IPar _ c1 c2) = ( c,
                                       (snd $ renameHidNames c1) ++
                                       (snd $ renameHidNames c2)
                                  )
renameHidNames c@(APar _ _ c1 c2) = ( c,
                                       (snd $ renameHidNames c1) ++
                                       (snd $ renameHidNames c2)
                                    )
renameHidNames c@(Ilv c1 c2) = ( c,
                                       (snd $ renameHidNames c1) ++
                                       (snd $ renameHidNames c2)
                               )
renameHidNames c@(Cond e c1 c2) = ( c,
                                       (snd $ renameHidNames c1) ++
                                       (snd $ renameHidNames c2)
                                  )
renameHidNames c@(Guard e circ) = (c,(snd $ renameHidNames circ))
renameHidNames c@(Mu _ circ) = (c,(snd $ renameHidNames circ))

rHN' circ hdn =
```

```
       let es = circNames circ
           ns = freshNames (es ++ hdn) (length hdn)
 in (circ,ns)


freshNames :: [String] -> Int -> [String]
freshNames ns 0 = []
freshNames ns k
 = let n' = freshName ns
   in  n' :  freshNames (n':ns) (k-1)
```

Name "Next" Actions

```
nmNextAct :: CircusProgram -> [(Circus, [CircusDef])]
nmNextAct defs
   = nNA defs
 where
      nNA [] = []
      nNA ((aname,(aparam,abody)):rest) = doNNA:nNA rest
        where
   doNNA = doNNA' abody aname

doNNA' :: Circus -> String -> (Circus, [CircusDef])
doNNA' circ aname
   = nNA' (circNames circ) aname circ

nNA' _ _ Div = (Div, [])
nNA' _ _ Stop = (Stop, [])
nNA' _ _ Skip = (Skip, [])
nNA' _ _ (nm := expr) = (nm := expr, [])

nNA' known aname circ@(Call n params)
  = (circ, [(n,(circNames circ,circ))])

nNA' known aname circ@(c1 ::: c2@(Call _ _)) = (c1' ::: c2, d')
  where (c1',d') = doNNA' c1 aname
nNA' known aname circ@(c1 ::: c2)
  = ( c1' ::: (Call nn [])
    ,  [(nn,([],c2'))] ++ d1' ++ d2'
    )
   where
     (c1',d1') = doNNA' c1 aname
     (c2',d2') = doNNA' c2 aname
     -- nn = freshNameAct (circNames c1'++circNames c2' ++ known)
     nn = aname
nNA' known aname circ@(IntChoice c1 c2)
   = (IntChoice (fst $ doNNA' c1 aname) (fst $ doNNA' c2 aname),
                (snd $ doNNA' c1 aname)++(snd $ doNNA' c2 aname))
nNA' known aname circ@(ExtChoice c1 c2)
   = (ExtChoice (fst $ doNNA' c1 aname) (fst $ doNNA' c2 aname),
                (snd $ doNNA' c1 aname)++(snd $ doNNA' c2 aname))
nNA' known aname circ@(Guard expr cp)
   = (Guard expr (fst $ doNNA' cp aname), (snd $ doNNA' cp aname))
nNA' known aname circ@(a :-> cp)
   = (a :-> (fst $ doNNA' cp aname), (snd $ doNNA' cp aname))
nNA' known aname circ@(Hide cp hdn)
   = (Hide (fst $ doNNA' cp aname) hdn, (snd $ doNNA' cp aname))
nNA' known aname circ@((a,as) ::-> cp)
   = ((a,as) ::-> (fst $ doNNA' cp aname), (snd $ doNNA' cp aname))
nNA' known aname circ@(evt :::-> cp)
   = (evt :::-> (fst $ doNNA' cp aname), (snd $ doNNA' cp aname))
nNA' known aname circ@(IPar a c1 c2)
   = (IPar a (fst $ doNNA' c1 aname) (fst $ doNNA' c2 aname),
```

```
             (snd $ doNNA' c1 aname)++(snd $ doNNA' c2 aname))
nNA' known aname circ@(APar a b c1 c2)
   = (APar a b (fst $ doNNA' c1 aname) (fst $ doNNA' c2 aname),
               (snd $ doNNA' c1 aname)++(snd $ doNNA' c2 aname))
nNA' known aname circ@(Ilv c1 c2)
   = (Ilv (fst $ doNNA' c1 aname) (fst $ doNNA' c2 aname),
          (snd $ doNNA' c1 aname)++(snd $ doNNA' c2 aname))
nNA' known aname circ@(Cond expr c1 c2)
   = (Cond expr (fst $ doNNA' c1 aname) (fst $ doNNA' c2 aname),
               (snd $ doNNA' c1 aname)++(snd $ doNNA' c2 aname))
nNA' known aname circ@(Mu nm cp)
   = (Mu nm (fst $ doNNA' cp aname), (snd $ doNNA' cp aname))
nNA' known aname circ@(INT nm a c)
   = (INT nm a (fst $ doNNA' c aname), (snd $ doNNA' c aname))
nNA' known aname circ@(SEQ nm a c)
   = (SEQ nm a (fst $ doNNA' c aname), (snd $ doNNA' c aname))
nNA' known aname circ@(EXT nm a c)
   = (EXT nm a (fst $ doNNA' c aname), (snd $ doNNA' c aname))
nNA' known aname circ@(IPAR a nm b c)
   = (IPAR a nm b (fst $ doNNA' c aname), (snd $ doNNA' c aname))
nNA' known aname circ@(APAR nm a b c)
   = (APAR nm a b (fst $ doNNA' c aname), (snd $ doNNA' c aname))
```

Get Variable Parameters

```
getCircVarParamsCalls :: CircusProgram
            -> [ ( String  -- definition name
                 , ( [String] -- variables used in definition (sorted)
                   , [String] -- actions called
                   )
                 )
               ]
getCircVarParamsCalls defs
 = alnorm $ gCVPC defs
 where
  gCVPC [] = []
  gCVPC ((aname,(aparam,abody)):rest)
   = (aname,(avars,acalls)):gCVPC rest
   where
    avars = lnorm $ collActionVarParams abody'
    acalls = lnorm $ collActionCalls abody'
    abody' = addContCall aname (circNames abody) (ensureAssgnCont (fst $ doNNA' abody aname))
```

```
collActionVarParams :: Circus -> [String]

collActionVarParams (x := e) = [x] ++ collExprVars e
collActionVarParams (Call act _) = []
```

```
collActionVarParams ((x,_) ::-> circ) = x : collActionVarParams circ

collActionVarParams (INT x _ circ) = x : collActionVarParams circ
collActionVarParams (SEQ x _ circ) = x : collActionVarParams circ
collActionVarParams (EXT x _ circ) = x : collActionVarParams circ
collActionVarParams (ILV x _ circ) = x : collActionVarParams circ
collActionVarParams (IPAR _ x _ circ) = x : collActionVarParams circ
collActionVarParams (APAR x _ _ circ) = x : collActionVarParams circ

collActionVarParams (a :-> circ) = collActionVarParams circ
collActionVarParams (c1 ::: c2) = collActionVarParams c1 ++
                                  collActionVarParams c2
collActionVarParams (IntChoice c1 c2) = collActionVarParams c1 ++
                                        collActionVarParams c2
collActionVarParams (ExtChoice c1 c2) = collActionVarParams c1 ++
                                        collActionVarParams c2
collActionVarParams (Hide circ _) = collActionVarParams circ
collActionVarParams (IPar _ c1 c2) = collActionVarParams c1 ++
                                     collActionVarParams c2
collActionVarParams (APar _ _ c1 c2) = collActionVarParams c1 ++
                                       collActionVarParams c2
collActionVarParams (Ilv c1 c2) = collActionVarParams c1 ++
                                  collActionVarParams c2
collActionVarParams (Cond e c1 c2) = collExprVars e ++
                                     collActionVarParams c1 ++
                                     collActionVarParams c2
collActionVarParams (Guard e circ) = collExprVars e ++
                                     collActionVarParams circ
collActionVarParams (Mu _ circ) = collActionVarParams circ
collActionVarParams _ = []
```

To gather information on the variables used in a particular expression, the function `collExprVars` is defined.

```
collExprVars (Var v) = [v]
collExprVars (Agg _ _ es) = concat $ map collExprVars es
collExprVars (App _ es) = concat $ map collExprVars es
collExprVars (Bin _ e1 e2) = collExprVars e1 ++ collExprVars e2
collExprVars _ = []
```

The function `collActionCalls` takes a particular Circus action and generates an array for gathering names of the calls to particular actions.

```
collActionCalls :: Circus -> [String]

collActionCalls (Call a _) = [a]
collActionCalls (n := e) = []

collActionCalls (c1 ::: c2) = collActionCalls c1 ++ collActionCalls c2
collActionCalls (a :-> circ) = collActionCalls circ
collActionCalls ((x,xs) ::-> circ) = collActionCalls circ
collActionCalls (IntChoice c1 c2) = collActionCalls c1 ++ collActionCalls c2
```

```
collActionCalls (ExtChoice c1 c2) = collActionCalls c1 ++ collActionCalls c2
collActionCalls (Hide circ a) = collActionCalls circ
collActionCalls (IPar a c1 c2) = collActionCalls c1 ++ collActionCalls c2
collActionCalls (APar a b c1 c2) = collActionCalls c1 ++ collActionCalls c2
collActionCalls (Ilv c1 c2) = collActionCalls c1 ++ collActionCalls c2
collActionCalls (Cond e c1 c2) = collActionCalls c1 ++ collActionCalls c2
collActionCalls (Guard e circ) = collActionCalls circ
collActionCalls (INT x a circ) = collActionCalls circ
collActionCalls (SEQ x a circ) = collActionCalls circ
collActionCalls (EXT x a circ) = collActionCalls circ
collActionCalls (ILV x a circ) = collActionCalls circ
collActionCalls (IPAR _ x _ circ) = collActionCalls circ
collActionCalls (APAR x _ _ circ) = collActionCalls circ
collActionCalls (Mu x circ) = collActionCalls circ \\ [x]
collActionCalls _ = []
```

The purpose of the `addAV` function is to attach the list of parameters to a Circus action. This is required because in the *CSP* world the variables of the *Circus* world turn into parameters and a particular action is invoked using parametric calls. While dealing with the assignment commands of the *Circus* world, the expression assigned to a particular variable is substituted in the parameter list.

```
addAV :: String -> [Expr] -> [String] -> Circus -> Circus
addAV nm plist calls (Call cnm pars) = (Call cnm (pars++plist))

addAV nm plist calls Skip = Skip

addAV nm plist calls (a :-> circ) = (a :-> (addAV nm plist calls circ))

addAV nm plist calls (c1 ::: c2) = (addAV nm plist calls c1 ::: addAV nm plist calls c2)

addAV nm plist calls (Guard e circ) = (Guard e (addAV nm plist calls circ))
addAV nm plist calls (IntChoice c1 c2) = (IntChoice (addAV nm plist calls c1)
                                                        (addAV nm plist calls c2))
addAV nm plist calls (ExtChoice c1 c2) = (ExtChoice (addAV nm plist calls c1)
                                                        (addAV nm plist calls c2))

addAV nm plist calls ((x,xs) ::-> circ) = ((x,xs) ::-> addAV nm plist calls circ)

addAV nm plist calls (Hide circ a) = (Hide (addAV nm plist calls circ) a)
addAV nm plist calls (IPar a c1 c2) = (IPar a (addAV nm plist calls c1)
                                                  (addAV nm plist calls c2))
addAV nm plist calls (APar a b c1 c2) = (APar a b (addAV nm plist calls c1)
                                                      (addAV nm plist calls c2))
addAV nm plist calls (Ilv c1 c2) = (Ilv (addAV nm plist calls c1)
                                            (addAV nm plist calls c2))
addAV nm plist calls (Cond e c1 c2) = (Cond e (addAV nm plist calls c1)
                                                  (addAV nm plist calls c2))
addAV nm plist calls (INT x a circ) = (INT x a (addAV nm plist calls circ))
addAV nm plist calls (SEQ x a circ) = (SEQ x a (addAV nm plist calls circ))
addAV nm plist calls (EXT x a circ) = (EXT x a (addAV nm plist calls circ))
addAV nm plist calls (ILV x a circ) = (ILV x a (addAV nm plist calls circ))
```

```
addAV nm plist calls (IPAR a x b circ) = (IPAR a x b (addAV nm plist calls circ))
addAV nm plist calls (APAR x a b circ) = (APAR x a b (addAV nm plist calls circ))
addAV nm plist calls body = body
```

Ensure Assignment Continutation

```
ensureAssgnCont :: Circus -> Circus

ensureAssgnCont Skip = Skip
ensureAssgnCont Stop = Stop
ensureAssgnCont Div = Div
ensureAssgnCont circ@(asg@(n := e) ::: c) = asg ::: ensureAssgnCont c
ensureAssgnCont circ@(Call a _) = circ

ensureAssgnCont (n := e) = (n := e) ::: Skip
ensureAssgnCont (c1 ::: c2) = ensureAssgnCont c1 ::: ensureAssgnCont c2
ensureAssgnCont (a :-> circ) = a :-> ensureAssgnCont circ
ensureAssgnCont ((x,xs) ::-> circ) = ((x,xs) ::-> ensureAssgnCont circ)
ensureAssgnCont (IntChoice c1 c2) = IntChoice (ensureAssgnCont c1) (ensureAssgnCont c2)
ensureAssgnCont (ExtChoice c1 c2) = ExtChoice (ensureAssgnCont c1) (ensureAssgnCont c2)
ensureAssgnCont (Hide circ a) = Hide (ensureAssgnCont circ) a
ensureAssgnCont (IPar a c1 c2) = IPar a (ensureAssgnCont c1) (ensureAssgnCont c2)
ensureAssgnCont (APar a b c1 c2) = APar a b (ensureAssgnCont c1) (ensureAssgnCont c2)
ensureAssgnCont (Ilv c1 c2) = Ilv (ensureAssgnCont c1) (ensureAssgnCont c2)
ensureAssgnCont (Cond e c1 c2) = Cond e (ensureAssgnCont c1) (ensureAssgnCont c2)
ensureAssgnCont (Guard e circ) = Guard e (ensureAssgnCont circ)
ensureAssgnCont (INT x a circ) = INT x a (ensureAssgnCont circ)
ensureAssgnCont (SEQ x a circ) = SEQ x a (ensureAssgnCont circ)
ensureAssgnCont (EXT x a circ) = EXT x a (ensureAssgnCont circ)
ensureAssgnCont (ILV x a circ) = ILV x a (ensureAssgnCont circ)
ensureAssgnCont (IPAR a x b circ) = IPAR a x b (ensureAssgnCont circ)
ensureAssgnCont (APAR x a b circ) = APAR x a b (ensureAssgnCont circ)
ensureAssgnCont (Mu x circ) = Mu x (ensureAssgnCont circ)
```

Add Continuation Calls

```
addContCall :: String -> [String] -> Circus -> Circus

addContCall _ _ Stop = Stop
addContCall _ _ Div = Div
addContCall _ _ circ@(n := e) = circ
addContCall _ _ circ@(Call a _) = circ

addContCall aname known Skip = Call aname []
```

```
addContCall aname known (c1 ::: c2) = (addContCall aname known c1) :::
                                      (addContCall aname known c2)
addContCall aname known (a :-> circ) = a :-> (addContCall aname known circ)
addContCall aname known ((x,xs) ::-> circ) = ((x,xs) ::-> (addContCall aname known circ))
addContCall aname known (IntChoice c1 c2) = IntChoice (addContCall aname known c1)
                                                      (addContCall aname known c2)
addContCall aname known (ExtChoice c1 c2) = ExtChoice (addContCall aname known c1)
                                                      (addContCall aname known c2)
addContCall aname known (Hide circ a) = Hide (addContCall aname known circ) a
addContCall aname known (IPar a c1 c2) = IPar a (addContCall aname known c1)
                                                (addContCall aname known c2)
addContCall aname known (APar a b c1 c2) = APar a b (addContCall aname known c1)
                                                    (addContCall aname known c2)
addContCall aname known (Ilv c1 c2) = Ilv (addContCall aname known c1)
                                          (addContCall aname known c2)
addContCall aname known (Cond e c1 c2) = Cond e (addContCall aname known c1)
                                                (addContCall aname known c2)
addContCall aname known (Guard e circ) = Guard e (addContCall aname known circ)
addContCall aname known (INT x a circ) = INT x a (addContCall aname known circ)
addContCall aname known (SEQ x a circ) = SEQ x a (addContCall aname known circ)
addContCall aname known (EXT x a circ) = EXT x a (addContCall aname known circ)
addContCall aname known (ILV x a circ) = ILV x a (addContCall aname known circ)
addContCall aname known (IPAR a x b circ) = IPAR a x b (addContCall aname known circ)
addContCall aname known (APAR x a b circ) = APAR x a b (addContCall aname known circ)
addContCall aname known (Mu x circ) = Mu x (addContCall aname known circ)
```

Propagate Assignment and Instantiate Continuation

```
propAssgn_InstCont :: String -> [Expr] -> [String] -> Circus -> Circus
propAssgn_InstCont nm plist calls (Call cnm pars) = (Call cnm (pars++plist))

-- when assignment is followed by a prefix e.g. x:=1;a:->Skip
propAssgn_InstCont nm plist calls ((v := expr) ::: (a :-> Skip))
        = ( a :-> (Call (foll_call calls nm) (map (esubstitute expr (Var v)) plist)))

-- when assignment is followed by a prefix to a call e.g. x:=1 ; a :-> Call "B"
propAssgn_InstCont nm plist calls ((v := expr) ::: (a :-> (Call cnm pars)))
        = ( a :-> (Call cnm (map (esubstitute expr (Var v)) plist)))

-- to deal with Guarded commanded with prefix to an assignment and a call
-- e.g. Guard e (a :-> v:=expr ; Call)
propAssgn_InstCont nm plist calls (Guard e (a :-> (v := expr):::(Call cnm pars)))
        = (Guard e (a :-> (Call cnm (map (esubstitute expr (Var v)) (pars++plist)))))

-- to deal with initialiser which has only assignments
propAssgn_InstCont nm plist calls ((v1 := expr1) ::: (v2 := expr2))
        = (Call (foll_call calls nm) (map (esubstitute expr1 (Var v1)) plist'))
        where plist' = (map (esubstitute expr2 (Var v2)) plist)
-- to deal with calls in sequence e.g. (Call "A" [] ::: Call "B" [])
propAssgn_InstCont nm plist calls ( (Call cnm1 pars1) ::: (Call cnm2 pars2))
        | nm == "MAIN" = Call cnm1 (pars1++pars2++plist)
        | otherwise = (Call cnm1 (pars1++plist)) ::: (Call cnm2 (pars2++plist))
```

```
-- when assignment is followed by a call e.g. x:=1;Call "a"
propAssgn_InstCont nm plist calls ((v := expr) ::: (Call cnm pars))
        | nm == "MAIN" =
            (Call (head calls)  (map (esubstitute expr (Var v)) (pars++plist)))
        | otherwise = (Call cnm (map (esubstitute expr (Var v)) (pars++plist)))

-- when assignment is followed by a call e.g. x:=1;Call "a";Call "b"
propAssgn_InstCont nm plist calls ((v := expr) ::: (Call cnm1 pars1) ::: (Call cnm2 pars2))
        | nm == "MAIN" =
            (Call (head calls)  (map (esubstitute expr (Var v)) (pars1++pars2++plist)))
        | otherwise = (Call cnm1 (map (esubstitute expr (Var v)) (pars1++plist)))

-- adding case for pA((A[]B);C) = (pA(A;C))[](pA(B;C))
propAssgn_InstCont nm plist calls ((ExtChoice c1 c2):::c3) =
                   (ExtChoice (propAssgn_InstCont nm plist calls (c1:::c2))
                              (propAssgn_InstCont nm plist calls (c2:::c3)))
-- adding case for pA((A|~|B);C) = (pA(A;C))|~|(pA(B;C))
propAssgn_InstCont nm plist calls ((IntChoice c1 c2):::c3) =
                   (IntChoice (propAssgn_InstCont nm plist calls (c1:::c2))
                              (propAssgn_InstCont nm plist calls (c2:::c3)))

propAssgn_InstCont nm plist calls Skip = Skip

propAssgn_InstCont nm plist calls (a :-> circ) =
      (a :-> (propAssgn_InstCont nm plist calls circ))


propAssgn_InstCont nm plist calls (c1 ::: c2)
       | nm == "MAIN" = propAssgn_InstCont nm plist calls c1
       | otherwise = (propAssgn_InstCont nm plist calls c1 :::
                      propAssgn_InstCont nm plist calls c2)


-- propAssgn_InstCont nm plist calls (v := expr) = (Call (foll_call calls nm)
                                       (map (esubstitute expr (Var v)) plist))

propAssgn_InstCont nm plist calls (Guard e circ) =
          (Guard e (propAssgn_InstCont nm plist calls circ))

propAssgn_InstCont nm plist calls (IntChoice c1 c2) =
          (IntChoice (propAssgn_InstCont nm plist calls c1)
                     (propAssgn_InstCont nm plist calls c2))
propAssgn_InstCont nm plist calls (ExtChoice c1 c2) =
          (ExtChoice (propAssgn_InstCont nm plist calls c1)
                     (propAssgn_InstCont nm plist calls c2))
propAssgn_InstCont nm plist calls ((x,xs) ::-> circ) =
          ((x,xs) ::-> propAssgn_InstCont nm plist calls circ)

propAssgn_InstCont nm plist calls (Hide circ a) =
          (Hide (propAssgn_InstCont nm plist calls circ) a)
propAssgn_InstCont nm plist calls (IPar a c1 c2) =
          (IPar a (propAssgn_InstCont nm plist calls c1)
                  (propAssgn_InstCont nm plist calls c2))
```

```
propAssgn_InstCont nm plist calls (APar a b c1 c2) =
          (APar a b (propAssgn_InstCont nm plist calls c1)
                    (propAssgn_InstCont nm plist calls c2))
propAssgn_InstCont nm plist calls (Ilv c1 c2) =
          (Ilv (propAssgn_InstCont nm plist calls c1)
                    (propAssgn_InstCont nm plist calls c2))
propAssgn_InstCont nm plist calls (Cond e c1 c2) =
          (Cond e (propAssgn_InstCont nm plist calls c1)
                  (propAssgn_InstCont nm plist calls c2))
propAssgn_InstCont nm plist calls (Guard e circ) =
          (Guard e (propAssgn_InstCont nm plist calls circ))

propAssgn_InstCont nm plist calls (INT x a circ) =
          (INT x a (propAssgn_InstCont nm plist calls circ))
propAssgn_InstCont nm plist calls (SEQ x a circ) =
          (SEQ x a (propAssgn_InstCont nm plist calls circ))
propAssgn_InstCont nm plist calls (EXT x a circ) =
          (EXT x a (propAssgn_InstCont nm plist calls circ))
propAssgn_InstCont nm plist calls (ILV x a circ) =
          (ILV x a (propAssgn_InstCont nm plist calls circ))
propAssgn_InstCont nm plist calls (IPAR a x b circ) =
          (IPAR a x b (propAssgn_InstCont nm plist calls circ))
propAssgn_InstCont nm plist calls (APAR x a b circ) =
          (APAR x a b (propAssgn_InstCont nm plist calls circ))
propAssgn_InstCont nm plist calls body = body


foll_call :: [String] -> String -> String
foll_call [] nm = (show Skip)
foll_call [call] nm = call++"_CONT"
foll_call (call:calls) nm
     | call == nm = head calls
     | otherwise = foll_call calls nm
```

Appendix D

Cache Coherence Protocol – Processes Specified using Simple*Circus*

The notation of Simple*Circus* developed for the SimpleCircus2CSPM tool is used to specify a cache coherence protocol given in [FS96]. This is included in Section 9.2, on page 89. The description of only two processes at the processor and memory is given there. Here, we include the remaining processes. The description of each of the process specified here is given on pages 3–6, [FS96].

Formal Description of Process p3:

(p3) *buf[p]?read_cache_freshR(q,r,cv,arg)* $\rightarrow$

$pred_p$:= q;
if r = *nil*
then $status_p$:=*Inlist*; cv_p:=*cv*; cs_p:=*fresh*
else *buf[r]!prependQ(p)*;$status_p$:=*Inqueue*;
if *arg* = *ok* **then** cv_p:=*cv*; cs_p:=*fresh* **fi fi**

```
cp6p3 = [
      ("P3"
          ,  ([]
            , (bufin "p" (read_cache_freshR "q,r,cv,arg")
              :::->
                 ( ("predProc" := (Val "q")) :::
                     (Cond (eq (Val "r") (Val "Nil"))
                        ( ("statusProc" := (Val "Inlist")) :::
                          ("cvProc" := (Val "cv")) :::
                          ("csProc" := (Val "Fresh"))
                        )

                        ( (bufout "r" (prependQ "p") :::-> Skip) :::
                          ("statusProc" := (Val "Inqueue")) :::
                          (Cond (eq (Var "arg") (Val "ok"))
                            (  ("cvProc" := (Val "cv")) :::
                               ("csProc" := (Val "Fresh"))
                            )
                            Skip
                          )
                        )
```

```
                        )
                    )
                )
            )
        )
    ]
```

Formal Description of Process p4:

(p4) *buf[p]?read_cache_goneR(q,r,cv,arg)* $\rightarrow$
pred$_p$:= q;
if r = *nil*
then *status*$_p$:=*Inlist*; *cv*$_p$:=*cv*; *cs*$_p$:=*dirty*
else *buf[r]!prependQ(p)*;*status*$_p$:=*Inqueue*;
if *arg* = *ok* **then** *cv*$_p$:=*cv*; *cs*$_p$:=*dirty* **fi fi**

```
cp6p4 = [
          ("P4"
          ,  ([]
             ,  (bufin "p" (read_cache_goneR "q,r,cv,arg")
                :::->
                ( ("predProc" := (Val "q")) :::
                  (Cond (eq (Val "r") (Val "Nil"))
                    (
                      ("statusProc" := (Val "Inlist")) :::
                      ("cvProc" := (Val "cv")) :::
                      ("csProc" := (Val "Dirty"))
                    )
                    ( bufout "r" (prependQ "p") :::-> Skip) :::
                      ("statusProc" := (Val "Inqueue")) :::
                      ( Cond (eq (Var "arg") (Val "ok"))
                        ( ("cvProc" := (Val "cv")) :::
                          ("csProc" := (Val "Dirty"))
                        )
                        Skip
                      )
                  )
                )
             )
          )
        )
    ]
```

Formal Description of Process p5:

(p5) *buf[p]?prependQ(q)* $\rightarrow$
if *status*$_p$ = *Inlist*
then *buf[q]!prependR(p,p,ok,cv*$_p$*,cs*$_p$*)*; *pred*$_p$:=q;
if *cs*$_p$ = *dirty* **then** *cs*$_p$:=*fresh* **fi**
else if *status*$_p$ = *Delleft*

then if $succ_p = nil$
then *buf[q]*!prependR(p,nil,ok,cv_p,cs_p);
cs_p:=*invalid*; $pred_p$:=*nil*
else *buf[q]*!prependR(p,$succ_p$,retry,cv_p,cs_p);
cs_p:=*invalid*; $pred_p$:=*nil* $succ_p$:=*nil* **fi**
else *buf[q]*!prependR(p,p,retry,cv_p,cs_p) **fi fi**

```
cp6p5 = [
        ( "P5"
        , ( []
          , (bufin "p" (prependQ "q")
            :::->
            ( Cond (eq (Var "statusProc") (Val "Inlist"))
               (
                 (bufout "q" (prependR "p,p,ok,cvProc,csProc"):::->Skip) :::
                 ("predProc" := (Val "q")) :::
                 (Cond (eq (Var "csProc") (Val "Dirty"))
                     ("csProc" := (Val "Fresh")) Skip)
               )

               (Cond (eq (Var "statusProc") (Val "Delleft"))
                   (
                     (Cond (eq (Var "succProc") (Val "nil"))
                       (
                         ( bufout "q"
                           (prependR "p,nil,ok,cvProc,csProc") :::-> Skip) :::
                         ("csProc" := (Val "invalid")) :::
                         ("predProc" := (Val "Nil"))
                       )

                       ( bufout "q"
                         (prependR "p,succProc,retry,cvProc,csProc") :::-> Skip) :::
                         ("csProc" := (Val "invalid")) :::
                         ("predProc" := (Val "Nil")) :::
                         ("succProc" := (Val "Nil"))
                       )
                    )

                    ( bufout "q" (prependR "p,p,retry,cvProc,csProc") :::-> Skip)
                 )
               )
            )
          )
        )
        ]
```

Formal Description of Process p6:

(p6) *buf[p]?prependR(q,r,arg,cv,cs)* $\rightarrow$
if *arg = ok*
then $status_p$:=*Inlist*; $succ_p$:=r;

if cs_p = *invalid* **then** cv_p:=*cv*; cs_p:=*cs*; **fi**
else *buf[r]*!*prependQ(p)* **fi**

```
cp6p6 = [
        ("P6"
        ,  ([]
           , (bufin "p" (prependR "q,r,arg,cv,cs"))
             :::->
             ( Cond (eq (Var "arg") (Val "ok"))
                 ( ("statusProc" := (Val "Inlist")) :::
                   ("succProc" := (Val "r")) :::
                   (Cond (eq (Var "csProc") (Val "invalid"))
                     (
                       ("cvProc" := (Val "cv")) :::
                       ("csProc" := (Val "cs"))
                     )
                     Skip
                   )
                 )
                 ( bufout "r" (prependQ "p") :::-> Skip )
             )
            )
        )
        ]
```

Formal Description of Process p7:

(p7) $status_p$ = *Inlist* ∧ cs_p = *dirty*
if $succ_p \neq nil$
then *buf[*$succ_p$*]*!*purgeQ(p)*; $status_p$:=*Purging*; $succ_p$:=*nil*
else cv_p:=? **fi**

```
cp6p7 = [
        ("P7"
        ,  ([]
           , (Guard (land (eq (Var "statusProc") (Val "Inlist"))
                          (eq (Var "csProc") (Val "Dirty"))))
                 (Cond (neq (Var "succProc") (Val "Nil"))
                     (
                        ( bufout "succProc" (purgeQ "p"):::-> Skip) :::
                        ("statusProc" := (Val "Purging")) :::
                        ("succProc" := (Val "Nil"))
                     )
                     (bufin "cvProc" "in" :::-> Skip)
                 )
           )
        )
        ]
```

Formal Description of Process p16:

(p16) *buf[p]?purgeQ(q)* $\rightarrow$
cs_p:=invalid; *buf[q]!purgeR(p,$succ_p$)*;
$pred_p$:=nil; *$succ_p$:=nil*;
if *$status_p$:=Inlist* **then** *$status_p$:=Off* **fi**

```
cp6p16 = [
         ("P16"
         , ([]
           ,  (bufin "p" (purgeQ "q"))
              :::->
              (
                 ("csProc" := (Val "invalid")) :::
                 (bufout "q" (purgeR "p,succProc"):::->Skip) :::
                 ("predProc" := (Val "Nil")) :::
                 ("succProc" := (Val "Nil")) :::
                 ( Cond (eq (Var "statusProc") (Val "Inlist"))
                    ("statusProc" := (Val "Off"))
                    Skip
                 )
              )
            )
         )
         ]
```

Formal Description of Process p17:

(p17) *buf[p]?purgeR(q,r)* $\rightarrow$
if r = *nil*
then *$status_p$:=Inlist*;*cv_p*:=?
else *buf[r]!purgeQ(p)* **fi**

```
cp6p17 = [
         ("P17"
         , ([]
           ,  (bufin "p" (purgeR "q,r"))
              :::->
              (Cond (eq (Var "r") (Val "Nil"))
                 ( ("statusProc" := (Val "Inlist")) :::
                   ((bufin "cvProc" "in"):::->Skip)
                 )
                 ( bufout "r" (purgeQ "p") :::-> Skip )
              )
           )
         )
         ]
```

Formal Description of Process p8:

(p8) $status_p$ = *Inlist* ∧ cs_p = *fresh* ∧ $pred_p$ = m →
buf[m]!*modifydataQ(p)*;$status_p$:=*Ftod*

```
cp6p8 = [
        ("P8"
        ,  ([]
           ,  (Guard (land
                        (land
                           (eq (Var "statusProc") (Val "Inlist"))
                           (eq (Var "csProc") (Val "Fresh"))
                        )
                        (eq (Var "predProc") (Val "m"))
                     )
              )
                (bufout "m" (modifydataQ "p"):::->Skip) :::
                ("statusProc" := (Val "Ftod"))
           )
        )
        ]
```

Formal Description of Process m4:

(m4) *buf[m]*?*modifydataQ(p)* →
if $head_m$ = p
then *buf[p]*?*modifydataR(m,ok)*;$staus_m$:=*Gone*
else *buf[p]*?*modifydataR(m,reject)*; **fi**

```
cp6m4 = [
        ("M4"
        ,  ([]
           ,  (bufin "m" (modifydataQ "p"))
              :::->
              ( Cond (eq (Var "headMem") (Val "p"))
                  ( (bufout "p" (modifydataR "m,ok"):::->Skip) :::
                     ("statusMem" := (Val "Gone"))
                  )
                  ( (bufout "p" (modifydataR "m,reject")):::->Skip)
              )
            )
        )
        ]
```

Formal Description of Process p9:

(p9) *buf[p]*?*modifydataR(q,arg)* →
$status_p$:=*Inlist*; **if** *arg* = *ok* **then** cs_p := *dirty* **fi**

```
cp6p9 = [
         ("P9"
         ,  ([]
            ,  ( (bufin "p" (modifydataR "q,arg"))
                 :::->
                 (("statusProc" := (Val "Inlist")) :::
                    (Cond (eq (Var "arg") (Val "ok"))
                      ("csProc" := (Val "Dirty"))
                      Skip
                    )
                 )
               )
            )
         )
         ]
```

(p10) $status_p = Inlist \wedge succ_p \neq nil \rightarrow$
$buf[succ_p]!delrightQ(p,pred_p,cs_p)$; $status_p$:= *Delright*

```
cp6p10 = [
          ("P10"
          , ([]
            , (Guard
                 (land
                    (eq (Var "statusProc") (Val "Inlist"))
                    (neq (Var "succProc") (Val "Nil"))
                 )
              )
              ( ((bufout "succProc" (delrightQ "p,predProc,csProc")):::->Skip) :::
                ("statusProc" := (Val "Delright"))
              )
            )
          )
          ]
```

Formal Description of Process p12:

(p12) *buf[p]?delrightQ(q,r,cs)* $\rightarrow$
if $status_p = Inlist \wedge pred_p$ = q
then *buf[q]?delrightR(p,ok)*; $pred_p$:= r;
if *cs* = *dirty* **then** cs_p := cs **fi**
else *buf[q]!delrightR(p,reject)*; **fi**

```
cp6p12 = [
          ("P12"
          , ([]
            , (bufin "p" (delrightQ "q,r,cs"))
              :::->
              ( Cond
                  (land (eq (Var "statusProc") (Val "Inlist"))
```

```
                    (eq (Var "predProc") (Val "q"))
                 )
                 (
                   ( (bufout "q" (delrightR "p,ok")):::->Skip) :::
                     ("predProc" := (Val "r")) :::
                     (Cond (eq (Var "cs") (Val "Dirty"))
                        ("csProc" := (Val "cs")) Skip
                     )
                 )
                 (  (bufout "q" (delrightR "p,reject")):::->Skip )
              )
            )
         )
         ]
```

Formal Description of Process p11:

(p11) $status_p = Inlist \wedge succ_p = nil \rightarrow$
$buf[pred_p]!delleftQ(p,nil,cv_p)$; $status_p := Delleft$

```
cp6p11 = [
         ("P11"
         , ([]
           ,  (Guard
                 (land (eq (Var "statusProc") (Val "Inlist"))
                       (eq (Var "succProc") (Val "Nil"))
                 )
              )
              (  ( (bufout "predProc" (delleftQ "p,nil,cvProc")):::->Skip) :::
                 ("statusProc" := (Val "Delleft"))
              )
           )
         )
         ]
```

Formal Description of Process p13:

(p13) $buf[p]?delrightR(q,arg) \rightarrow$
if $cs_p = invalid$
then $status_p = Off$
else if $arg = reject$
then $status_p = Inlist$
else $buf[pred_p]!delleftQ(p,succ_p,cv_p)$;
$status_p := Delleft$ **fi fi**

```
cp6p13 = [
```

```
("P13"
, ([]
  , (bufin "p" (delrightR "q,arg"))
    :::->
    (Cond (eq (Var "csProc") (Val "invalid"))
        ("statusProc" := (Val "Off"))
        ( Cond (eq (Var "arg") (Val "reject"))
             ("statusProc" := (Val "Inlist"))
             ((bufout "predProc" (delleftQ "p,succProc,cvProc")):::->Skip)
                ::: ("statusProc" := (Val "Delleft"))
        )
    )
  )
)
]
```

Formal Description of Process m3:

(m3) *buf[m]?delleftQ(p,q,cv)* $\rightarrow$
if $head_m$ = p
then cv_m:=*cv*;*buf[p]*!*delleftR(m,ok)*; $head_m$:= q;
if q = *nil* **then** $status_m$:=*Home* **fi**
else *buf[p]*!*delleftR(m,reject)*; **fi**

```
cp6m3=[
  ("M3"
      , ([]
        , (bufin "m" (delleftQ "p,q,cv"))
          :::->
          ( Cond (eq (Var "headMem") (Val "p"))
               (
                  ("cvMem" := (Val "cv")) :::
                  (bufout "p" (delleftR "m,ok"):::->Skip) :::
                  ("headMem" := (Val "q")):::
                  (Cond (eq (Var "q") (Val "Nil"))
                       ("statusMem" := (Val "Home"))
                       Skip
                  )
               )
               (bufout "p" (delleftR "m,reject"):::->Skip)
          )
        )
      )
      ]
```

Formal Description of Process p14:

(p14) *buf[p]?delleftQ(q,r,cv)* $\rightarrow$
if $succ_p$ = q $\wedge$ ($status_p$ = *Inlist* $\vee$ $status_p$ = *Ftod*

$\vee$ $status_p$ = *Delright*)
then *buf[q]*!*delleftR(p,ok)*; $succ_p$:=r
else *buf[q]*!*delleftR(p,reject)*; **fi**

```
cp6p14 = [
         ("P14"
         , ([]
           , (bufout "p" (delleftQ "q,r,cv")
              :::->
              (Cond (land
                       (eq (Var "succProc") (Val "q"))
                       (lor
                          (lor
                             (eq (Var "statusProc") (Val "Inlist"))
                             (eq (Var "statusProc") (Val "Ftod"))
                          )
                          (eq (Var "statusProc") (Val "Delright"))
                       )
                    )
                  (
                     (bufout "q" (delleftR "p,ok"):::->Skip) :::
                     ("succProc" := (Val "r"))
                  )
                  ( bufout "q" (delleftR "p,reject"):::->Skip)
               )
             )
           )
         )
         ]
```

Formal Description of Process p15:

(p15) *buf[p]*?*delleftR(q,arg)* $\rightarrow$
if cs_p = *invalid* $\vee$ *arg* = *ok*
then $succ_p$ = *nil*; $pred_p$ = *nil*; $status_p$ = *Off*;
else *buf[*$pred_p$*]*!*delleftQ(p,*$succ_p$*,*cv_p*)* **fi**

```
cp6p15 = [
         ("P15"
         , ([]
           ,  ( bufin "p" (delleftQ "q,r,cv")
                :::->
                (Cond (lor
                         (eq (Var "csProc") (Val "invalid"))
                         (eq (Var "arg") (Val "ok"))
                      )
                )
                (
                   ("succProc" := (Val "Nil")) :::
```

```
            ("predProc" := (Val "Nil")) :::
            ("csProc" := (Val "invalid")) :::
            ("statusProc" := (Val "Off"))
         )
         ( bufout "q" (delleftR "p,reject") :::-> Skip)
       )
    )
  )
  ]
```

Appendix E

How to Run Examples in SimpleCircus2CSPM and circus2cspm

E.1 Haskell Implementation of Translator

The development of this translator is done using GHCi (Haskell compiler / interpreter) version 6.2.2.
The working machine had a Core2Duo processor running Windows 7 OS.

E.1.1 Steps to Follow

1. Download GHCi 6.2.2 version.
2. Install the Haskell compiler / interpreter.
3. Get the SimpleCircus2CSPM .lhs files from the given URL:
   `https://www.scss.tcd.ie/~begm/finalWork.html`
4. Run Cmd (command prompt).
5. Go to the folder where the .lhs files of SimpleCircus2CSPM tool are placed.
6. Type `ghci` and press Enter.
7. Run command `:load MainSimpleCircus.lhs` (This will load all necessary files to the compiler)
8. Now you can run the examples. The command to run each example is mentioned in the Evaluation chapter where the output of each example is discussed.

E.2 Java Implementation of Translator – circus2cspm

The development of this version is done using Eclipse IDE.
The working machine had a Core2Duo processor running Windows 7 OS.

E.2.1 Steps to Follow

1. Download circus2cspm archive from the given URL `https://www.scss.tcd.ie/~begm/finalWork.html`
2. Download LaTeX files from the *Circus* examples' archive.
3. Unzip the folders where you want.
4. You should have the appropriate JDK version installed on your machine. (The version we have is JRE version 6).
5. Go to the circus2cspm unzipped folder.
6. Run the Windows Batch File named `Circus2cspm`.
7. This will start the GUI of the circus2cspm tool.
8. Now specify the path of the *Circus* LaTeX file in the "Input Specification" field.
9. The project name has to be specified.
10. Specify the path where you want to create the folder of CSP_M output files.
11. Hit the "Translate" Button.