Universidade Federal do Rio Grande do Norte

A Strategy to verify the Code Generation from concurrent and state-rich *Circus* specifications to Executable Code

Samuel Lincoln Magalhães Barrocas

PhD thesis

Natal - RN 2018

Universidade Federal do Rio Grande do Norte

A Strategy to verify the Code Generation from concurrent and state-rich *Circus* specifications to Executable Code

Samuel Lincoln Magalhães Barrocas
Supervisor: Marcel Vinícius Medeiros Oliveira
Co-supervisor: Jim Woodcock

PhD thesis

Thesis presented to the PPgSC as a requirement for the acquisition of a PhD degree
Concentration field: Software Engineering/Formal Methods

Natal - RN 2018

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Barrocas, Samuel Lincoln Magalhães.
A strategy to verify the code generation from concurrent and state-rich circus specifications to executable code / Samuel Lincoln Magalhães Barrocas. - 2018.
234f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Departamento de Informática e Matemática Aplicada, Programa de Pós-Graduação em Sistemas e Computação. Natal, RN, 2018.
Orientador: Marcel Vinícius Medeiros Oliveira.
Coorientador: Jim Woodcock.

1. Engenharia de software - Tese. 2. Métodos formais - Tese. 3. Verificação de código - Tese. 4. Testes de software - Tese. 5. Síntese de código - Tese. 6. Circus - Tese. I. Oliveira, Marcel Vinícius Medeiros. II. Woodcock, Jim. III. Título.

RN/UF/CCET CDU 004.41

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

SAMUEL LINCOLN MAGALHÃES BARROCAS

Uma Estratégia para Validar a Geração de Códigos de Circus para Java

Esta Tese foi julgada adequada para a obtenção do título de doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Prof. Dr. Marcel Vinicius Medeiros Oliveira – UFRN
Orientador

Prof. Dr. Bruno Motta de Carvalho – UFRN
Coordenador do Programa

Banca Examinadora

Prof. Dr. Marcel Vinicius Medeiros Oliveira – UFRN
Presidente

Prof. Dr. Martin Alejandro Musicante – UFRN
Examinador

Prof. Dr. Umberto Souza da Costa – UFRN
Examinador

Prof. Dr. Alexandre Cabral Mota – UFPE
Examinador

Prof. Dr. Emerson Gurgel Gomes – IFRN
Examinador

Fevereiro, 2018

Contents

Resumo

O uso de Geradores Automáticos de Código para Métodos Formais não apenas minimiza esforços na implementação de Sistemas de Software, como também reduz a chance da existência de erros na execução destes Sistemas. Estas ferramentas, no entanto, podem ter faltas em seus códigos-fonte que causam erros na geração dos Sistemas de Software, e então a verificação de tais ferramentas é encorajada. Esta tese de Doutorado visa criar e desenvolver uma estratégia para verificar JCircus, um Gerador Automático de Código de um amplo sub-conjunto de *Circus* para Java. O interesse em *Circus* vem do fato de que ele permite a especificação dos aspectos concorrentes e de estado de um Sistema de maneira direta. A estratégia de verificação consiste nos seguintes passos: (1) extensão da Semântica Operacional de Woodcock e prova de que ela é sólida com respeito à Semântica Denotacional existente de *Circus* na Teoria Unificada de Programação (UTP), que é um framework que permite prova e unificação entre diferentes teorias; (2) desenvolvimento e implementação de uma estratégia que verifica o refinamento do código **gerado** por JCircus, através de uma toolchain que engloba um Gerador de Sistema de Transições Rotuladas com Predicado (LPTS) para *Circus* e um Gerador de Modelos que aceita como entrada (I) o LPTS e (II) o código **gerado** por ***JCircus***, e gera um modelo em Java Pathfinder que verifica o refinamento do código **gerado** por ***JCircus***. Através da aplicação do passo (2) combinada com técnicas baseadas em cobertura no código **fonte** de ***JCircus***, nós visamos aumentar a confiabilidade do código **gerado** de *Circus* para Java.

Palavras-chave: Métodos Formais, verificação de código, Testes de Software, Síntese de código, *Circus*.

Abstract

The use of Automatic Code Generators for Formal Methods not only minimizes efforts on the implementation of Software Systems, but also reduces the chance of existing errors on the execution of such Systems. These tools, however, can themselves have faults on their source codes that may cause errors on the generation of Software Systems, and thus verification of such tools is encouraged. This PhD thesis aims at creating and developing a strategy to verify the code generation from the *Circus* formal method to Java Code. The interest in *Circus* comes from the fact that it allows the specification of concurrent and state-rich aspects of a System in a straightforward manner. The code generation envisaged to be verified is performed by ***JCircus***, a tool that translates a large subset of *Circus* to Java code that implements the JCSP API. The strategy of verification consists on the following steps: (1) extension of Woodcock's Operational Semantics to *Circus* processes and proof that it is sound with respect to the Denotational Semantics of *Circus* in the Unifying Theories of Programming (UTP), that is a framework that allows proof and unification of different theories; (2) development and implementation of a strategy that refinement-checks the code **generated** by ***JCircus***, through a toolchain that encompasses (2.1) a Labelled Predicate Transition System (LPTS) Generator for *Circus* and (2.2) a Model Generator that inputs (I) a LPTS and (II) the code **generated** by ***JCircus***, and generates a model (that uses the Java Pathfinder code model-checker) that refinement-checks the code **generated** by ***JCircus***. Combined with coverage-based techniques on the **source** code of ***JCircus***, we envisage improving the reliability of the Code Generation from *Circus* to Java.

Keywords: Formal Methods, Model-Checking, Software Testing, Code Synthesis, *Circus*.

Chapter 1

Introduction

On this introduction, we will firstly expose concepts about Formal Methods, and then expose the motivation of the work, the contributions and the scope of the PhD thesis.

1.1 Formal Methods

Formal Methods are techniques for specifying Systems using mathematical constructs. They allow the reasoning about the properties of a System that must be satisfied, envisaging not only a better understanding about its conditions and states but also to pursuit its correct behaviour and prevent errors on the System. Examples of Formal Methods are: π-Calculus (Robin Milner, Joachim Parrow, David Walker 1990), CSP (Hoare 1985), Z (Woodcock & Davies 1996), VDM (Bear 1988), ASM (Börger & Stärk 2003), Actor Model (Carl Hewitt .), B (Abrial 1996), among others.

Differently from other reasoning techniques, for Formal Methods a System is reasoned exclusively by its states, behaviours and conditions that must be satisfied on each state (like pre-conditions and post-conditions) and by how its concurrent parts (that are called processes) interact. A state, in Formal Methods, can be represented by the values of the variables of its corresponding state space. State changes can be represented both as equality expressions between the future state (where variables are marked with a dash ', for example: x') and the current state of each variable. For example, let's suppose that there are two states with variables $x : \mathbb{N}$, $y : \mathbb{N}$ and $z : \mathbb{N}$ as follows:

State 1 : $x = 5 \land y = 7 \land z = 8$

State 2 : $x = 8 \land y = 7 \land z = 8$

The State Change between states 1 and 2 can be seen as follows:

State Change : $x' = x + 3 \wedge y' = y \wedge z' = z$

In Concurrent Systems, states and state changes are implicit (not easy to notice). That is why is relevant for them to have Operational Semantics (Hoare 1985, Freitas 2006, For 1999). The Operational Semantics of a concurrent formalism has rules to construct a Labelled Transition System (LTS) for the Formal Method. The LTS explicits the state space of the specification by treating it as a State Machine with the computational steps until reaching their final states.

Approaches of formal methods encompass Process Algebras (in which CSP and π-Calculus are based) and model-based approaches (in the case of B, Z and VDM, for example). Process Algebras provide constructs for the explicit specification of Concurrent (Parallel) Systems. In Concurrent systems, there are processes being executed in parallel (that is, at the same time) that can interact with each other throught channels. Channels are static entities that allow synchronisation between processes. The synchronisation can transfer a value from one process to the other, and when that happens, the synchronisation is called communication. Both synchronisation and communication are activities performed on the channel, and we call this an event. There are states where a process P can possibly perform different events and another process Q synchronise on one of the events offered by P. If a set of events is offered by a process, it is said that this process offers an External Choice between those events. There are also states where a process can either offer two different events, and it is not known what event will be offered: in this case, we say that there is a non-determinism of events being offered. Non-determinism is introduced throught the Internal Choice construct. Both π-Calculus and CSP have similar primitives, with CSP being more complete and sofisticated on specifying External Choices and non-determinism. Both of them, however, lack constructs for storing values on variables and for specifying state changes in a straight manner.

The Model-based approach is suitable for specifying Systems where it is relevant expliciting the states and state changes of the program. The Z-notation, for example, allows state changes throught structures called schemas. B, on the other hand, allows state changes through operations that have pre and post-conditions inside a B-machine with an invariant.

The Model-based approach and the Process Algebras can have their semantics compared and unified in the Unifying Theories of Programming (UTP) (Hoare & Jifeng 1998), that is a logic theory that describes formal constructs as First Order Logic predicates. These predicates encompass special variables, called observational variables. The nature of the

UTP as a logic theory allows any Formal Method that has a theory on the UTP has each of its constructs described as an UTP predicate, and this allows properties to be proved: it is throught UTP that is possible to link a Denotational Semantics to an Operational Semantics.

Both the Model-based approach and the Process Algebras have limitations on their approaches. This motivated researchers to combine formalisms of both approaches in order to increase the power of the specifications. There were combinations of Z with CCS (Galloway & Stoddart 1997), Z with CSP (Fischer 1998), B with CSP (Butler & Leuschel 2005), among others. These combinations, however, allowed formalisms to be used only separately. The power of formalism can be even bigger when notations are mixed throughout the specification. With this motivation, the *Circus* notation was created (Woodcock & Cavalcanti 2001).

Circus is a combination of three notations: Z, CSP and Dijkstra's Guarded Commands (Dijkstra 1975). *Circus* also has a Refinement Calculus (Oliveira 2006*a*) that allows the specification to be syntactically changed without semantically changing it. As CSP, *Circus* allows the creation of processes. The difference between both notations is that a *Circus* process can carry information, inside its state. The Operational Semantics of *Circus* was firstly described in (Freitas 2006), having its rules defined by Z schemas, and this motivated the implementation of a model-checker prototype. Another version of the Operational Semantics of *Circus* was produced in (J. C. P. Woodcock 2007), and the last release of an Operational Semantics for a formalism that encompasses constructs that lie on *Circus* is given in (Cavalcanti & Woodcock 2013). *Circus* also has a Denotational Semantics (Oliveira 2006*a*) that gives definitions for each construct of *Circus* in terms of their pre and post-conditions.

1.2 Motivation

Although Formal Methods provide reasoning about Software correctness, the implementation of Software from Formal Methods may not be free from faults. A fault is a wrong code implementation that has an error on the execution of the code as a consequence. This happens because there can be errors on the specification during the process of manual implementation. Moreover, the manual implementation of Software from Formal Methods tend to be costly. An interesting effort not only for reducing the odds of errors but also for reducing the costs of the implementation from the formal specification is the use of Automatic Code Generators for Formal Methods: with this purpose, Code Generators for several Formal Methods were created: from the B method to executable

code (Bossu & Requet 2000, Mammar & Laleau 2006), from Event-B to ADA (Edmunds et al. 2012), from Event-B to C (Wright 2009, Méry & Singh 2011, Méry & Singh 2010), from Event-B to SPARK (Murali & Ireland 2012, Gkatzia 2011), from π-Calculus to Java (Li 2005*b*, Formiga & Lins 2009), from π-Calculus to .NET (Li 2005*a*), from Z to Java (Miyazawa 2008), CSP to both Java (to CTJ (Hilderink et al. 1997) and to JCSP (Welch et al. 2010)) and C (CCSP) (V. Raju & Stiles 2003, Raju et al. 2003), from LOTOS to PARLOG (Gilbert 1988), from Perfect to C, C++ and Java (Perfect Developer) (Crocker 2003), from Esterel to Java (Fekete & Richard 1998), among others. Related to the strategy from *Circus* to Java, (Oliveira 2006*a*) firstly developed a translation strategy from *Circus* to Java that uses a Java library that implements some CSP constructs, called JCSP (Welch 2000, Welch et al. 2010). This strategy was implemented on the first version of ***JCircus*** (Freitas 2005) and had many limitations due to the lack of support of CSP constructs in JCSP. Some of them were resolved on (Barrocas 2011) and (Barrocas & Oliveira 2012), but others remained. ***JCircus*** allows the automatic generation of Java code for *Circus* specifications with a Graphical User Interface (which is also generated by the strategy) that permits the interaction with the code of the process, allowing the interaction between the user and the code generated.

Although Automatic Code Generators for Formal Methods may reduce both the costs of development of Software from formal specifications and the odds of error, they also do not prevent errors completely. That happens because there can be errors on the object-code (generated by the Automatic Code Generator) caused by faults on the source code of the Automatic Code Generator. Efforts on strenghening the reliability of the translation strategy encompass both: (1) a stepwise and proved-to-be-correct compilation (possibly with refinement steps) from the specification until reaching the code generated (Abrial 1996, Palsberg 1992*b*, Palsberg 1992*a*, Duran et al. 2010, Sampaio 1997, Leuschel 2008, Leroy 2008, Leroy 2009) and (2) a mixed approach where Formal Verification and Software Testing are combined (Garavel & Sifakis 1990, Júnior 2016, Kennedy et al. 2013, Dantas et al. 2009, Aljer et al. 2003, Evans & Grant 2008, de Matos & Moreira 2012), envisaging detecting faults on the source code of the Compiler/Code Generator and increasing the reliability of the object-code that the Code Generator generates. The strategy we adopted on this PhD thesis combines ideas from Refinement/Model-Checking, Software Testing and Source Code Coverage, envisaging detecting faults on our Automatic Code Generator.

1.3 Contributions of this work

1.3.1 A full and sound Operational Semantics for *Circus*

Circus has two levels of constructs: the level of processes and the level of actions (these are activities that can be performed by a *Circus* process). For both levels, all compound operators (those operators that have two operands, like External Choice, Internal Choice, Parallelism and etc) exist both for *Circus* actions and for *Circus* processes. The most elementary process of *Circus* is the basic process: it has a state that can be specified as a Z-schema and a set of actions with a mandatory main action, that corresponds to the behaviour of the process. Compound processes are processes that are defined in terms of two or more processes. Unary processes are processes whose behaviour is defined by a single process and whose behaviour can be changed by another construct (for example: a Rename Process is defined by a given process and pairs between an old name and a new name of a channel).

The Operational Semantics described by (Freitas 2006, Cavalcanti & Woodcock 2013, Cavalcanti & Gaudel 2011) defines rules only for some *Circus* actions and basic processes. As the translatable grammar of ***JCircus*** encompasses compound processes (processes that are defined through two or more processes), we need an Operational Semantics that has rules for all the constructs of *Circus*. Oliveira (Oliveira 2006*a*) created refinement laws for *Circus* actions, basic processes and some kinds of compound and unary processes (processes that have only one process as a branch), in which some of them (1) allow the transformation of a compound process between basic processes into a single basic process and (2) the transformation of an unary process with a basic process into a single basic process. We first considered the hypothesis of creating rules for *Circus* processes using the existent kinds of transitions on (Cavalcanti & Woodcock 2013, Cavalcanti & Gaudel 2011), but as their design was for *Circus* actions (where there is not an encapsulated state), their use for processes would be unfeasible.

We gave, on this thesis, a theoretical contribution to the works of Oliveira, Freitas, Woodcock, Cavalcanti and Gaudel (Oliveira 2006*a*, Freitas 2006, Cavalcanti & Woodcock 2013, Cavalcanti & Gaudel 2011) by creating a set of rules that describes a stepwise syntactic transformation from **any** *Circus* composition into a basic process. This set of rules was joint to Woodcock's set of rules and formed a full Operational Semantics for *Circus*.

The role of the proofs is to have a theory that is sound (proved correct) with respect to the Unifying Theories of Programming (UTP) for the Operational Semantics, and thus

strenghening the implementation of the Labelled Predicate Transition System Compiler for *Circus*. More information about the results on the soundness of the Operational Semantics of *Circus* can be found at (Barrocas & Oliveira 2017).

1.3.2 A Labelled Predicate Transition System Compiler for *Circus*

A Labelled Transition System (LTS) is a graph-like structure that represents the computational steps of a program. Each vertice of this graph is called node and is the remaining program to be executed, and each edge is an arc that takes the program to a source node to a destination node. A Labelled Predicate Transition System (LPTS) is an abstraction of the LTS where communicated values and algebraic expressions are stored, in each node, as symbolic loose constants that have the value of its corresponding expression.

The first prototype of an LPTS Compiler for *Circus* was developed by Freitas (Freitas 2006). This prototype was developed before the release of the first version of the parser of *Circus* and used a mix of different Abstract Syntax Trees. Moreover, it was developed, according to Freitas, in a time when there was no full grammar for *Circus*.

The second contribution we gave, on this thesis, is the implementation of an LPTS Compiler for *Circus*, based on the Operational Semantics for *Circus* (whose soundness we will prove on this PhD thesis) we quoted on 1.3.1. More details about the LPTS Compiler are given on section 5.2.1.

The role of the LPTS Compiler is to provide, for any *Circus* specification, a graph-like structure (LPTS) that represents the stepwise computation of that specification. The LPTS Compiler is a tool of the toolchain we provided on this thesis. The other tool for the toolchain is the JPF Model Generator, that will be described on the following sub-section.

1.3.3 A Refinement-Checking Strategy between Model and Code

Refinement-checking tools, like FDR (For 1999), checks if a model X is refined by model Y by calculating semantic equivalence between the Labelled Transition System (LTS) of each model. FDR calculates if the LTSs of both models are bisimilar. Bisimilarity is given by checking that:

$$\forall (m, n) : Node\ x\ Node \land (m', n') : Node\ x\ Node \,.\, (m \xrightarrow{l} m') \Rightarrow (n \xrightarrow{l} n') \qquad (1.1)$$

Formula 1.1 checks if: for all pairs of states in which each state belongs to an LTS, if there is an arc labelled l going out from a given source node to a given destination node on the

first LTS, then there is the same arc *l* going out from a source node to a destination node on the second LTS.

The semantic-equivalence checking described above requires both LTSs as inputs. Nevertheless, we created a technique to check semantic equivalence between a model and a code. On our case, if the LPTS of *Circus* is semantically equivalent to a Java code (generated by ***JCircus***) that implements the *Circus* specification. The advantage of our technique, compared to the one used on FDR, is that it does not require both LPTSs for calculating the refinement-checking. The semantic equivalence between the specification and the code generated is calculated by automatically simulating successive clicks on the GUI of the code generated by ***JCircus***, guided by the LPTS, and checking the responses. The technique described on this paragraph was implemented as the JPF Model Generator, a tool that inputs both a Labelled Predicate Transition System (LPTS) and a JCSP code (generated by ***JCircus***), and outputs a model that is able to refinement-check the JCSP code generated by ***JCircus***. Figure 1.1 illustrates this difference.

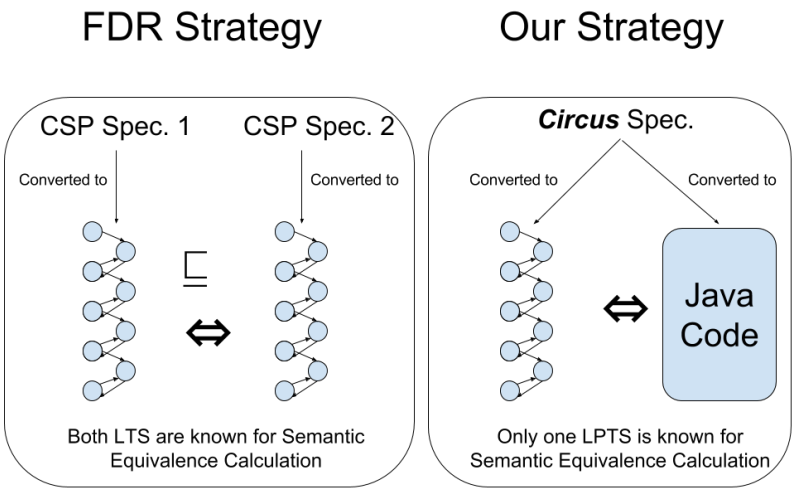

Figure 1.1: Differences between FDR and Our Strategy

On Figure 1.1, it is possible to see that FDR receives two specifications as input (**CSP Spec. 1** and **CSP Spec. 2**) for a refinement-checking. Both **CSP Spec. 1** and **CSP Spec. 2** are converted to Labelled Transition Systems and they are compared using formula 1.1. If the left LTS contains the right LTS, then **CSP. Spec 1** is refined ($\sqsubseteq$) by **CSP. Spec 2**. If both **CSP. Spec 1** is refined ($\sqsubseteq$) by **CSP. Spec 2** and **CSP. Spec 2** is refined ($\sqsubseteq$) by **CSP. Spec 1**, then there is an equivalence ($\Leftrightarrow$) between both LTS, and thus, between both CSP processes. Our strategy, however, focus on only one *Circus* specification (*Circus* **Spec.**)

and generates both the LPTS (from the LPTS Generator) and Java code (from ***JCircus***), checking if both are semantically equivalent. The Semantic Equivalence calculation will be performed by a JPF Model that is generated from the JPF Model Generator (it takes both the LPTS and the Java code generated by ***JCircus*** as input).

The semantic-equivalence we created and implemented for this thesis can be calculated in two levels of completeness: (1) one in which the code generated by ***JCircus*** is semantically tested verifying what it does, and (2) another in which the code generated by ***JCircus*** is semantically tested verifying what it does and does not do. The advantage of the former is a lower cost of processing, comparing to the latter, and the disadvantage of the former is a lower completeness. What the system does is determined by sequences called traces, whereas what it does not do is determined by sets called refusals.

Works that have similarity with ours are Gaudel and Cavalcanti's testing theory for *Circus* (Cavalcanti & Gaudel 2011, Cavalcanti & Gaudel 2010), based on symbolic traces that serve as a basis for the construction of test cases for a *Circus* System Under Test (SUT). These works, however, differently from our work, do not perform test of failures (only of traces), as also do not deal with non-determinism (a situation that can be caused, for example, by Internal Choice).

1.4 Objectives of this work

The goal of this PhD thesis is to create and develop a technique for verifying the code generation from concurrent and state-rich specifications in *Circus* to Java Executable Code. The core of our technique is ***JCircus***, the translator from *Circus* to Java. To achieve that, we need to verify the code **generated** by ***JCircus*** and check if it is equivalent to the semantics of the input specification. Moreover, this was applied to an input set that guarantees that the **source** code of ***JCircus*** is properly covered. The more covered the source code of ***JCircus*** is, the more accurate is the verification. The enumeration of steps of this document is:

1. Soundness of the Operational Semantics for *Circus* with respect to the UTP (Hoare & Jifeng 1998): a sound Operational Semantics has a set of rules that describe a correct sequence of computational steps. Thus, the role of these proofs is to prove that the Operational Semantics that will serve as a basis to the LPTS Compiler is sound with respect to the Denotational Semantics of *Circus* (Oliveira 2006*a*). Providing formal evidence that the Operational Semantics is correct strenghtens the process of verification. We show, on this document, partial results concerning these proofs on

the anexes from C.2 to C.3.21;

2. Implementation of an LPTS Compiler for *Circus*, based on the Operational Semantics quoted on the previous item;

3. Implementation of a Model Generator that model-checks the code generated by ***JCircus***: this module will take as input both (1) a Transition System returned by the LPTS Compiler and (2) the code **generated** by ***JCircus*** and generate a Java Pathfinder (JPF) model that will interact with the code generated by ***JCircus*** in a similar manner the user interacts with the code generated by ***JCircus***, by automatically conducting clicks on buttons, but checking if the results of the interaction correspond or not to a successfull refinement. The model is going to use the Java Pathfinder (NASA 2005) model checker;

4. Application of the previous steps to an input set that guarantees high ***JCircus*** source code coverage: the previous set of steps allows the application of a model-checking strategy to the code generated by ***JCircus*** in order to check if it refines the semantics of the input. If this is applied to a set of inputs that reaches high **source** code coverage, odds of finding errors on the code generation from *Circus* to Java increases. The percentage of Code Coverage will be measured by EclEmma (Hoffmann 2006);

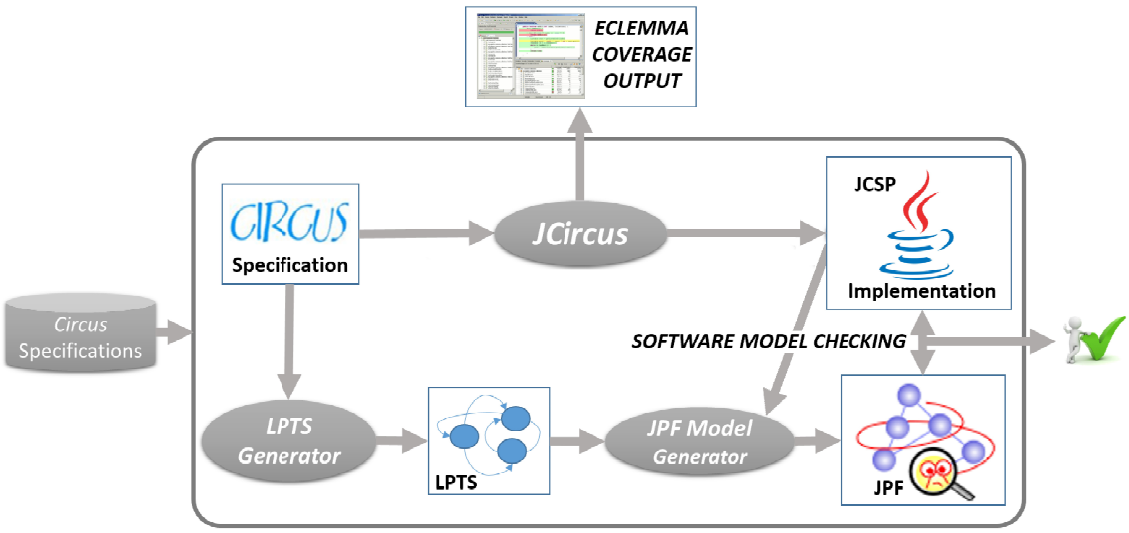

Figure 1.2: Strategy for verifying ***JCircus***

Figure 1.2 shows the full scope of this PhD thesis related to tool implementation (this figure alongside the contribution described in subsection 1.3.1 form the full scope). On figure 1.2,

there is an input set of specifications in *Circus* (*Circus* **Specifications**) that is submitted to ***JCircus*** and the toolchain. Each *Circus* **specification** is submitted as input to the **LPTS Generator** and to ***JCircus***. ***JCircus***, then, translates the specification into JCSP code (**JCSP Implementation**). When submitted as input to the **LPTS Generator**, there will be a conversion of the specification to an **LPTS** that, alongside **JCSP Implementation**, will be the input of the **JPF Model Generator**. The output of the JPF Model Generator is a **JPF** model that refinement-checks the **JCSP Implementation**. It is expected, as a verification step, that this refinement-checking is successfull. The process is repeated until it is found, on the source code of ***JCircus***, a satisfiable coverage from **Eclemma Coverage Output**.

This document is organized as follows:

- Chapter 2 shows related work and we present a discussion between our work and these works;
- Chapter 3 explains the *Circus* notation, and the explanation is based on an example of an Air Controller and its grammar. Then it explains *Circus*' Operational Semantics and Denotational Semantics;
- Chapter 4 explains ***JCircus*** and its basic contents: JCSP (the API of the code generated by ***JCircus***), CZT (Community Z Tools, the framework that implements the parser of *Circus* and its typechecker), LaTeX *Circus* (the syntax of *Circus* that ***JCircus*** accepts as input), and the structure of ***JCircus***;
- Chapter 5 explains the process used to verify ***JCircus***. It starts with the basic contents (Java Pathfinder and EclEmma) and then explains the implementation of the LPTS Compiler and the JPF Model Generator. Then it shows information about the inputs and the code coverage achieved;
- Chapter 6 shows our conclusions and future work;
- The Anexes that start with C.2 and finish with C.3.21 attaches the Operational Semantics laws presented on chapter 3 and show proofs of part of the rules from the Operational Semantics of *Circus* that was used on this document;

On the next chapter, we will discuss related work.

Chapter 2

Related Work

On this chapter, we discuss Related Work for Code Generators, their limitations with respect to verification and correctness, comparing with our work.

2.1 Formal Methods and Code Generation

There are a number of works that deal with Code Generation for Formal Methods, both for Hardware and Software programming languages. Among Code Generation for Software programming languages, there are works related to the generation from the B method to executable code (Bossu & Requet 2000, Mammar & Laleau 2006), from Event-B to ADA (Edmunds et al. 2012), from Event-B to C (Wright 2009, Méry & Singh 2011, Méry & Singh 2010), from Event-B to SPARK (Murali & Ireland 2012, Gkatzia 2011), from π-Calculus to Java (Li 2005*b*, Formiga & Lins 2009), from π-Calculus to .NET (Li 2005*a*), from Z to Java (Miyazawa 2008), CSP to both Java ((Hilderink et al. 1997, Welch et al. 2010)) and C (V. Raju & Stiles 2003, Raju et al. 2003), from LOTOS to PARLOG (Gilbert 1988), from Perfect to C, C++ and Java (Crocker 2003), from Esterel to Java (Fekete & Richard 1998) and previous works on the Code Generation from *Circus* to Java (Oliveira 2006*a*, Freitas 2005, Freitas & Cavalcanti 2006, Barrocas 2011, Barrocas & Oliveira 2012, Barrocas & Oliveira 2016).

On the next section, we will discuss works on code generation between a formal method and a programming language.

2.1.1 Code Generation between a Formal Method and a Programming Language

Code Generation for Software from B-based formalisms (B and Event-B) encompasses database applications (in the case of (Mammar & Laleau 2006)) and embedded applications

(in the case of (Bossu & Requet 2000), where there is a translation from B to C code that can be embedded on smartcards).

For π-Calculus, (Li 2005*a*) presents an approach for manually implementing π-Calculus processes in .NET using an intermediate step that produces an Operational Semantics of π-Calculus. The Operational Semantics is also presented as a contribution of (Li 2005*a*). Other work that deals with π-Calculus is (Formiga & Lins 2009), that presents an approach for manually implementing π-Calculus in Java Virtual Machine Bytecode in an efficient manner. The role of this work was to serve as a basis for the creation of concurrent programming languages based on the π-Calculus.

The notion of a stepwise refinement prior to the translation is present both in (Wright 2009), where there is a semantic conversion of an abstract Event-B specification into a concrete specification that can be easily translated to the C programming language, and in (Freitas 2005, Freitas & Cavalcanti 2006, Barrocas 2011, Barrocas & Oliveira 2012), where an abstract (and non-translatable) *Circus* specification is converted to a concrete (translatable) specification.

Similarities with these works and ours rely on the stepwise refinement prior to the code generation itself (in the case of our work, the Refiner module applies refinement laws in order to prepare the specification for the Parallelism Updater module) and on the use of an Operational Semantics as an intermediate step. The difference is the role of the Operational Semantics: in (Li 2005*a*), it is used as an intermediate step for **translating** π-Calculus into .NET, whereas in our strategy it is used as a step for **verifying a translation**. Moreover, no verification approach is known for (Li 2005*a*).

(Raju et al. 2003) presents tools that automatically convert a CSP subset to Java code or C code. The Java generated code is both in CTJ and in JCSP. The C generated code uses the CCSP library. The motivation of the paper is to reduce the occurrence of errors in the implementation of the specifications into executable code, and reduce the effort on the implementation of these specifications. In the paper, there are several examples specified in CSP translated to JCSP, CTJ and CCSP. These examples involve the translation of processes, alphabetized parallelism (only in the cases in which the processes synchronise on the events that they have in common), external choice, communication (only with one field), and prefixing. Our strategy has many advantages comparing to (Raju et al. 2003): (1) we generate code from specifications written in a more complex formalism (*Circus* encompass the syntaxes of Z and CSP) into Java code, (2) we generate code for communications with arbitrary number of fields and field decorations, and (3) we have verified the code generation from *Circus* to Java, whereas on (Raju et al. 2003), no verification approach is known. A minor disadvantage between our work and (Raju

et al. 2003) is the lack of support for libraries CTJ and CCSP.

In (Edmunds et al. 2012), the authors describe, using an example, the addition of new types to a formal translator, in order to extend a translation strategy from Event-B to Ada. The author of the paper created new translation rules for Event-B types to Ada. An advantage of our work comparing to this work is the existence of a verification strategy of the code generated by ***JCircus***.

In (Murali & Ireland 2012, Gkatzia 2011), the authors describe an approach for translating Event-B specifications to the SPARK programming language, in order to answer how properties defined in a specification language can be translated into provably correct code. The proof of correctness of the generated code can be assured by producing a semantics of the program code. The project encompassed the development of the E-Spark tool, which is an Event-B plug-in for the generation of Event-B to SPARK Ada. The authors took an advantage of generating code from Event-B to a formally defined programming language, enabling the use of pre and post-conditions and, thus, easing the process of generating correct code, whereas our verification approach did not reach correctness.

For Hardware programming languages, there are Code Generators from CSP to Handel-C (Phillips & Stiles 2004, Medeiros Junior 2012, Oliveira, Júnior & Woodcock 2013, Jr et al. 2012, Macário & Oliveira 2015), from B to LLVM (Júnior 2016) (that uses the BETA test generator (de Matos & Moreira 2012)) , among others. The works from CSP to Handel-C had advancements on the translation of CSP processes in the programming of FPGAs: support for multi-synchronisation through a protocol and support for Hiding (Jr et al. 2012). Our strategy did not have to use a multi-synchronisation protocol since (Welch et al. 2010), that provided partial support for multi-synchronisation through alting barriers. Related to the Hiding construct, our support is partial: only those Hiding expressions whose Hiding can be ignored are translated (see item 4 of section 4.4.2), whereas in (Jr et al. 2012), no restrictions on Hiding translation exist. Our strategy, thus, has a verification approach that is not present on (Phillips & Stiles 2004, Medeiros Junior 2012, Oliveira, Júnior & Woodcock 2013, Jr et al. 2012, Macário & Oliveira 2015).

Gutemberg (Júnior 2016) extends the regular B-method verification approach (with Atelier-B (ClearSy 2003)) to the level of assembly, increasing the reliability of the verification. It uses Cid's (de Matos & Moreira 2012) BETA B-based test generator to generate test cases for LLVM. LLVM is a Compiler Infrastructure that optimizes compilation and execution of programs written in diverse compilers. A comparison between (Júnior 2016) and our work is shown at 2.3.

2.1.2 Code Generation between a Formal Method and a Formal Method

Other works focus on generating formal methods from formal methods (or from modeling languages). A straghtforward difference between our work and this group of works is the nature of the output: here, the output is a Formal Method, whereas in our approach the output is an code written in the Java programming language. Although they differ from our work in the nature of the Generated Code , this is an important group of works because they allow a straightforward integration of all tools from the formalism being generated to the source formalism. E.g.: VDM to B (Bicarregui et al. 2000) enables Atelier-B to be used for VDM since VDM is converted to B. The works are: Object-Z to Perfect Developer (Kimber 2007), Rebeca to mCRL (Hojjat et al. 2007), LOTOS to Petri-nets (Garavel & Sifakis 1990), RSL to CSP (Vargas et al. 2008), *Circus* to CSP (Oliveira et al. 2014), among others.

(Kimber 2007) developed a translator of formal specifications written in Object-Z for the Perfect language, thus enabling verification of the specification translated in Perfect Developer. Perfect Developer is a model-checking tool that uses the Perfect language, which implements Verified Design By Contract, an extension of the Design By Contract system introduced by Eiffel. This work performs this translation through the formal mapping of Object-Z to the Perfect language. Similarities between our approach and Kimber's (Kimber 2007) approach is the lack of support for some constructs: it does not support (1) sets and types defined on-the-fly by the user in Object-Z, (2) Inheritance, (3) Generic-types and their constructs (formal parameters and actual calls), (4) Iterated Operators, (5) Renaming, (6) Variable Hiding, and (7) Class Hierarchy. Disregarding all object-oriented constructs from Object-Z, the other constructs are common to the Z part of *Circus*. As our support for the Z part is restricted to freetypes and a given format of axiomatic definition, our approach is more limited than Kimber's approach for these constructs. On the other hand, we provide code generation for a large subset of the CSP part of *Circus* and, thus, for specifying concurrent systems (something that cannot be done straighforwardly in Object-Z). Considering verification, the mapping from Object-Z to Perfect, as far as we know, is not verified, whereas our approach had a verification of the code generation strategy.

(Oliveira et al. 2014) presents a theoretical mapping between *Circus* and CSP constructs, through functions that transform a (possibly state-rich) *Circus* process into a CSP process (necessarily stateless, as CSP does not store data). This work makes an important contribution because it can link *Circus* to existing tools for working with CSP (FDR included). An important information about this work is that the mapping from *Circus* to

CSP is sound with respect to the Unifying Theories of Programming (UTP), allowing the correct use of *Circus* on all CSP tools (including FDR (For 1999)). This work, if combined with ours, can strongly alleviate the lack of support for constructs on the code generation from *Circus* to Java (as most CSP constructs from *Circus* are translated, whereas support for Z constructs is very limited).

(Bicarregui et al. 2000) describes a partial translation from the formal language VDM to the B-method. VDM (Vienna Development Method) is a formal language based on First Order Logic, in which functions and state transformation operations can be implicitly determined, with pre and post-conditions, or explicitly. This approach has neither a verification approach developed nor a proof of correctness from VDM to B. Our approach, on the other hand, has a Software Testing verification approach.

(Hojjat et al. 2007) presents a mapping from the Rebeca (Reactive Object Language) formal language to the mCRL2 formal language, and this mapping is implemented in a tool called Sarir, that charges this translation automatically. A Rebeca model contains a set of concurrent objects, in which each object is a ”rebec” (reactive object). Common to our work is the nature of the input formalism: Rebeca, alongside *Circus*, allows modelling of concurrent objects. Differently from our approach, however, is the full support from Rebeca to mCRL2 (no non-supported constructs are reported on the paper), whereas our strategy do not support major part of Z in *Circus*. Moreover, no verification approach was detected on this work, differently from our approach.

(Boudiaf & Djebbar 2009) proposes an automatic translation of CPN to the language Maude. CPN is an acronym for Colored Petri-Nets. CPN is a graphical language whose goal is to model and validate distributed and concurrent systems. Maude is a specification language based on rewriting logic (this is a logic that allows transformations on a predicate without changing its semantics). The Maude System allows graphical editing and simulating Colored Petri Nets. The tool allows drawing a CPN in a graphic fashion and the automatic translation of the graphical representation of the CPN to Maude specification. The process from CPN to its graphic representation, showing its states, looks like our process of transforming the *Circus* specification into its Labelled-Predicate Transition System (LPTS). The simulation phase can be compared with the execution with the code generated by ***JCircus***: when executing the code generated by ***JCircus***, we are simulating the specification. No non-translated constructs were reported on (Boudiaf & Djebbar 2009), whereas ***JCircus*** do not support major part of Z constructs. The Maude System, however, allows the verification of the Coloured Petri Nets, whereas our strategy verifies ***JCircus***.

(Tripakis et al. 2005) presents a method for translating discrete-time Simulink models to the Lustre language. The authors of the paper aimed at having a tool in which models

were designed in Simulink, translated to Lustre, and implemented on a platform that used the Lustre C code generator and a C compiler for this platform. Simulink is an environment for block-diagram modelling of systems and signals. Lustre is a programming language designed to implement reactive systems. The method of translation is implemented in a tool called S2L. Similar to our work is the support for only a subset of the input formalism: only the discrete-time part of Simulink is translatable, (the continuous part is not translatable). No correctness or verification approach was found on the strategy from Simulink to Lustre, whereas our strategy performs verification of the code generation from *Circus* to Java.

There are also approaches that deal with manual verified construction or automatic verified generation of code generators. When they are automatic, they are called Compiler Generators. Examples of these approaches are the works of Palsberg (Palsberg 1992*b*, Palsberg 1992*a*), Leroy (Leroy 2008, Leroy 2009, Leroy 2014, Leroy 2016), Sampaio (Sampaio 1997) and Duran (Duran et al. 2010).

2.2 Code Generators with limited support

Many of these works have limitations on their translation strategies due to lack of constructs of the formal language on their target codes. Both works that deal with code generation from CSP to Java (V. Raju & Stiles 2003, Raju et al. 2003) have limitations on the generation of Communication, Multi-synchronisation and General Interleaving. Previous works on the translation from *Circus* to Java (Freitas 2005, Freitas & Cavalcanti 2006) had partially overcome the limitation on Multi-synchronisation by providing a verified Multi-Synchronisation protocol. This protocol was later dispensed on (Barrocas 2011, Barrocas & Oliveira 2012), where there was an integration of ***JCircus*** with the newest version of JCSP at the time. The other limitations were partially overcome on (Barrocas 2011, Barrocas & Oliveira 2012): it was provided support for Communication with arbitrary sequence of fields (either input, output or dot) and for General Interleaving. Another work that had limitations on its translation strategy was CSP to Handel-C (Phillips & Stiles 2004). Some limitations of (Phillips & Stiles 2004) were overcome by Medeiros (Medeiros Junior 2012, Oliveira, Júnior & Woodcock 2013, Jr et al. 2012): this work allowed the definition of local processes, the renaming of channels, the use of boolean guards in external choices, and also implemented a communication protocol that eliminated some restrictions on the parallel composition of processes in the translation to Handel-C. Another limitation that was resolved was Channel Abstraction (Hiding), on paper (Macário & Oliveira 2015).

The mapping from Object-Z to Perfect developer (Crocker 2003) does not cover all

the constructs from Object-Z. From the 88 constructs from the grammar of Object-Z, 61 are fully supported. The constructs with partial or lack of support encompass: Inheritance, Types, Generic Parameters, Iterated Operators, Renaming and Variable Hiding. The author did not mention why those constructs were not mapped to Perfect: it only said that the mapping was not possible to be done. Nevertheless, the author warned that the inheritance mapping is feasible, since Object-Z has inheritance.

Other works that have limited support are:

- (Li 2005*b*, Formiga & Lins 2009) did not treat controlled operations;
- (Tripakis et al. 2005) translates only the discrete-part of Simulink, that is, only blocks of the "Discrete" Library and generic mathematical operators, such as sum;
- The tool described in (Murali & Ireland 2012) covers only the sequential part of Event-B, not covering, for example, concurrent systems;
- Axiomatic Descriptions and Infinite Data Types are not handled by (Miyazawa 2008);

On sections 2.3 and 2.4 we will expose the code generators that had some verification applied to them. We will divide in: (1) Verified Approaches with Software Testing Techniques, and (2) Verified Approaches of Correctness.

2.3 Verification Approaches with Software Testing Techniques

On this section, we will detail related work on Code Verification with Software Testing Techniques. These works generally apply test suites (manually constructed or automatically generated by tools) as Oracles to generated codes by Formal Code Generators, indicating if the generated code satisfies its input specification. They also generally apply their techniques to an input set with proper Code Coverage in order to increase the reliability of the verification.

From the quoted works, Gutemberg (Júnior 2016) refer to formal verification of these Code Generators using Software Testing Techniques. This work do not provide correctness of its generated code, but its formal approach increases the reliability of its generated code. Its approach has a good similarity with our approach: Gutemberg uses Cid's BETA tool (de Matos & Moreira 2012) as a test case generator whose generated test suit works as an Oracle that decides wether the generated code has similar semantics to the input specification, and using Code Coverage Measure as a part of the strategy. Gutemberg (Júnior 2016) provides a verification strategy from B to the LLVM assembler (Lattner 2002).

This strategy uses test cases (the test Oracles) that are generated by the BETA tool (de Matos & Moreira 2012) for each B specification and apply the generated code for B to these test cases. In our case, the toolchain (LPTS Compiler + JPF Model Generator) plays the same role as that of BETA on (Júnior 2016). The difference is that, instead of generating test case sets, the Oracle generated by our toolchain is a JPF Model that refinement-checks the generated code by ***JCircus*** and determines if the generated code has a similar semantics to the input specification. As non-determinism is translated as a random choice in Java, we needed a Model/Refinement-checker as an Oracle. Another difference is that Cid and Gutemberg generate the test suite directly from the Abstract Syntax Tree of the B specification, whereas our approach generates the Operational Semantics of the specification (the LPTS) as an intermediate step between the *Circus* specification and the JPF Model (an example of work that uses the Semantics as an intermediate step is (Nogueira et al. 2008), that describes an approach for test generation from CSP models guided by the traces of the model). The approach of (Júnior 2016, de Matos & Moreira 2012) is under continuation by (de Azevedo Oliveira 2017), enforcing the verification strategy by applying it to more general coverage criteria, like MC/DC.

Merry's and Singh's EB2C (Méry & Singh 2011, Méry & Singh 2010) is another example of Formal Code Generator that was verified using Code Verification and Software Testing. The approach used by both authors, however, used a meta-proof that proved that the abstract execution of the C generated code satisfies the execution of the Event-B input specification. The authors say that they have guaranteed correctness of EB2C, however, they have applied their strategy to an input set that satisfied proper coverage criteria from the source code of EB2C. So, we classify this work as a "Code Verification and Software Testing" work. The similarity of this work with ours is the type of input (a Formal Method) and the output (a code in a programming language) and the use of Coverage based techniques on the verification.

If we consider, on this review, Verification Approaches without automatized code generation or manual code implementation, we can include other works. The work of (Mandrioli et al. 1995) presents an approach for generating functional (black-box) test cases for real-time systems, having the formal specification of these systems (written in the TRIO formalism) as input. A difference between this work and ours is the fact that the System Under Test, in the case of (Mandrioli et al. 1995), was not automatically generated from the specification. A similarity between our strategy and that of (Mandrioli et al. 1995) is that, in the case of finite systems, it is able to perform all possible testing simulations (a simulation, in the context of the paper, is a combination of input values) from the formal input properties to the Real-time System Under Test in the case it has a finite number

of possible behaviours. In our case, our model-checking strategy is limited for infinite data types: a range of integer values must be determined before model-checking starts. The problem, however, of exercising all paths of the System Under Test (which are not necessarily exercised when all posible paths of the formal model from which the test cases were generated are exercised), is handled by the approach of (Liu & Nakajima 2011). (Liu & Nakajima 2011) presents a method called "Vibration", that has as a starting point a test condition from a test scenario. "Vibration" traverses paths of the System Under Test collecting information about the paths traversed and continues to traverse while the test condition is satisfied. Our refinement-checking approach also performs a traversal on the code generated by ***JCircus*** intercepting all random instructions (generated by internal choices) and controlling the path traversed by a value such that all non-deterministic paths are traversed.

On a theoretical reasoning, the works of Gaudel and Cavalcanti (Cavalcanti & Gaudel 2011, Cavalcanti & Gaudel 2010) develop a Testing theory for *Circus* based on symbolic traces. The idea is to submit a System Under Test (SUT) to a test suit that is built from an Operational Semantics of *Circus*. That work differs from ours firstly on not using an Operational Semantics that has rules for processes, so they assume firstly that the Formal Specification that is used as the basis for testing the SUT is already refined on a Basic Process. Moreover, their strategy does not test refusals (while ours, as a model-checking strategy from model to code, does). Another difference is that their theory does not assure that the Test Suite exercises all possible paths of the SUT, due to possible non-determinism on the SUT, whereas our refinement-checking strategy described at 5.2.2 does (the role of the System Under Test, on our case, is played by the code generated by ***JCircus***). The approach of (Cavalcanti & Gaudel 2011, Cavalcanti & Gaudel 2010) was recently extended on (Alberto et al. 2017) by abstracting the approach in order to create mutants. The approach follows a similar reasoning of that used on Mutation testing, but creating Formal Specification Mutants, that are used to evaluate the quality of the symbolic traces that compose the Test Suite: if the traces break the tests on the mutant, it is killed. If all mutants are killed, then the symbolic traces that compose the Test Suite are ok.

2.4 Verification Approaches of Correctness

Works from the approach of correctness of code generation encompass Leroy (Leroy 2008, Leroy 2009, Leroy 2014, Leroy 2016), Palsberg (Palsberg 1992*b*, Palsberg 1992*a*), Sampaio (Sampaio 1997), and Duran (Duran et al. 2010). These works present Formal Code Generators that both (1) were manually implemented using rigorous techniques and

whose refinement steps until the object/generated-code were proved correct, or (2) were automatically generated using Compiler Generators (that will be also considered on this section).

Palsberg's Cantor (Palsberg 1992*b*, Palsberg 1992*a*) is a correct Compiler Generator that inputs an Action Semantics description and outputs a proved-to-be-correct Compiler that translates programs written on the language represented by the Action Semantics into assembly code. It was primarily used for generating ADA compilers (Palsberg 1992*a*), but it can also be used for formal methods since the Action Semantics description of that formal method is formalised. The proof of correctness of the Compiler Generator was made by converting the Action Semantics into the machine semantics using Horn clauses (Horn 1951).

Leroy's CompCert (Leroy 2008, Leroy 2009, Leroy 2014, Leroy 2016) is a proved-to-be-correct Compiler for the C language. Compcert inputs a big subset of C, which they call Clight, and outputs PowerPC assembly code. The proof of correctness was made by transforming the formal semantics of Clight into the formal semantics of PowerPC. This transformation, according to (Leroy 2009), is divided in 14 steps. From them, some are not verified: the parser, type-checker, the simplifier that generates the abstract syntax of Clight, a printer for Abstract Syntax Trees in concrete Assembly Syntax, and the generation of executable binary using the system's assembler and linker. In comparison to our work, the refiner step of ***JCircus*** (explained at 4.4) had its manual implementation based on refinement laws that were proved correct, and the parallelism updater step (also explained at 4.4) had its implementation justified by a FDR model that was proved correct (this model lies on (Barrocas 2011)). The other steps were not based on correct models.

Finally, the works of Sampaio (Sampaio 1997) and Duran (Duran et al. 2010) show algebraic approaches for the construction of correct compilers for programming languages. The difference between both is that Sampaio's approach is for imperative languages, whereas Duran's approach is for Object Oriented Compilers with Copy Semantics, that is, for OO Compilers without pointers. Sampaio's approach could be used to formal notations like B or Z with proper adaptation, since it was designed for imperative programming languages.

2.5 Summary

On this section, we will sumarize the related works to ours (including ours) that had tools implemented on the following table.

Input and Output	Tools	Some related works
Circus to Java (JCSP)	***JCircus***	(Barrocas & Oliveira 2012)
CSP to Java and C	-	(Raju et al. 2003)
Event-B to C	EB2C	(Wright, Mery & Singh 2010, 2011)
Event-B to Ada	A Rodin plug-in	(Edmunds et al. 2012)
Event-B to SPARK	E-SPARK	(Murali & Ireland 2012)
π-Calculus to Java	pi2j	(Li 2005*b*, Formiga & Lins 2009)
from π-Calculus to .NET	pi-calculator	(Li 2005*a*)
Rebeca to mCRL2	Sarir	(Hojjat et al. 2007)
Simulink to Lustre	S2L	(Tripakis et al. 2005)
Petri-nets do Maude	Maude System	(Boudiaf & Djebbar 2009)
Action Semantics to Machine Code	Cantor	(Palsberg 1992*b*, Palsberg 1992*a*)
Clight to PowerPC	Compcert	(Leroy 2008, 2009, 2014, 2016)
ROOL to RVM	-	(Duran et al. 2010)
Ext.Dijkstra's to Mac. Code	-	(Sampaio 1997)
Perfect to C, C++ and Java	Perfect Developer	(Crocker 2003)
LOTOS to PARLOG	lopar	(Gilbert 1988)
Esterel to Java	Unknown name	(Fekete et al. 1998)
B to LLVM	B2LLVM	(Júnior 2016)
B to C	Atelier-B	(ClearSy 2003)
Circus to CSP	-	(Oliveira et al. 2014)
Object-Z to Perfect	OZIFY 1.0	(Kimber 2007)
VDM to B	Unknown name	(Bicarregui et al. 2000)
Z to Java	JZed Plug-in	(Miyazawa 2008)

On table 5.2.3, we can see that some approaches do not have tools for them: these are (Raju et al. 2003, Duran et al. 2010, Sampaio 1997, Oliveira et al. 2014). In the case of (Duran et al. 2010, Sampaio 1997, Oliveira et al. 2014), however, there are proofs that their stepwise refinements are correct, whereas no proof is provided on (Raju et al. 2003).

Chapter 3

Circus and its Semantics

3.1 *Circus*

Concurrent and Integrated Refinement CalculUS (*Circus*) (Woodcock & Cavalcanti 2001, Freitas 2005, Freitas 2006, Oliveira 2006*a*) is a formal language whose syntax is based on the sintaxes of two other formal languages, Z (Woodcock & Davies 1996) and CSP (M. G. Hinchey and S. A. Jarvis 1995). *Circus* joins Z's feature of representing complex data structures with CSP's process algebra, that represents concurrency. *Circus* also has a refinement calculus (Oliveira 2006*a*), and its syntax encompasses Dijkstra's language of guarded commands (Dijkstra 1975).

Like in CSP, a *Circus* program is formed by one or more processes, each of which can be run in Parallel. Differently from CSP, however, a *Circus* process has a state that can carry inner information about it. As a consequence of possibly having inner information, *Circus* has two levels of constructs: the level of processes and the level of actions. An action can be seen as a piece of behaviour that can be used by the *Circus* process. Processes themselves can also be a piece of behaviour of other processes, and, as a consequence, many operators of *Circus* are overloaded: they can be used both for processes and for actions.

Circus processes and actions can synchronise with each other through channels. Channels are static entities on a *Circus* program that allows synchronisation and communication of values. For example: process P performs a synchronisation on **channel** c and then skips. In this case, the text may refer to the fact that P performs **event** c and then skips.

Each channel can have one or more fields, in which values are communicated. The nature of the communication can be of input (?), output (!) or dot (.). When there is an input (?) communication like $c?x$ synchronising with an output communication $c!5$ or a dot communication $c.5$, then there is a transfer of value between $c!5$ and $c?x$ such that x stores 5 as the value communicated.

We will exemplify *Circus* with an example: it is a room controller that regulates its temperature according to the preference of the user (Barrocas & Oliveira 2016). The system has three processes: *CONTROLLER*, *SENSOR* and *AIRCONTROLLER*. We will show each process and explain one by one, but starting from the freetypes declared, *STPLUG* and *STTURN*.

$$STPLUG ::= IN \mid OUT$$
$$STTURN ::= ON \mid OFF$$

STPLUG indicates if the system is plugged or not (*IN* for plugged, and *OUT* for unplugged). *STTURN* is a freetype that indicates if the system is turned on or not (*ON* for turned on, and *OFF* for turned off). Having explained the freetypes used by the processes of the specification, now we will explain the *CONTROLLER* process.

$$\begin{array}{l}
\textbf{process}\ CONTROLLER\ \widehat{=}\ \textbf{begin} \\
\quad \textbf{state} ACST\ ==\ [preferred : \mathbb{N}] \\
\quad INIT\ \widehat{=}\ preferred := 25 \\
\quad CTR\ \widehat{=}\ \mu X \bullet \left(\begin{array}{l}
switchoff \rightarrow Skip \\
\Box\ preferredtemp?np \rightarrow preferred := np\ ;\ X \\
\Box\ startcycle \rightarrow getplug?p \rightarrow getturn?t \rightarrow gettemp?tp \rightarrow \\
\qquad \left(\begin{array}{l}
(p = IN \wedge t = ON \wedge preferred < tp)\ \& \\
\qquad cooldown!preferred \rightarrow endcycle \rightarrow X \\
\Box\ (p = IN \wedge t = ON \wedge tp \leq preferred)\ \& \\
\qquad cooldown!tp \rightarrow endcycle \rightarrow X \\
\Box\ (p \neq IN \vee t \neq ON)\ \& \\
\qquad endcycle \rightarrow X
\end{array} \right)
\end{array} \right) \\
\quad \bullet\ INIT\ ;\ CTR \\
\textbf{end}
\end{array}$$

The *CONTROLLER* process has a single state component, called *ACST*, that is formed by the schema $[preferred : \mathbb{N}]$, that declares natural variable *preferred*, for storing the preferred temperature by the user. Furthermore, this process has two actions, *INIT* and *CTR*. Action *INIT* initialises the variable preferred to 25 ($preferred := 25$). Action *CTR* is more sofisticated: it has a fixed point (μX) in which are offered external choices ($\Box$) between (1) switching off the controller (*switchoff*) and then terminating (*Skip*), or (2) setting the preferred temperature (*preferredtemp*?*np*) having then the value of *preferred*

being updated ($preferred := np$) sequenced (;) by the action recursing to the X fixed point (μX), or (3) starting a cycle (*startcycle*) for cooling down the room. If a cycle is started, the controller then retrieves the status of the plug (*getplug*?p), then the status of the sensor (*getturn*?t), and then the environment's temperature (*gettemp*?tp). Then , if the plug is in ($p = IN$) and the system is turned on ($t = ON$) and the preferred temperature is less than the captured temperature ($preferred < tp$), it cools down the room by outputting the preferred temperature (*cooldown*!*preferred*), then it indicates the end of the cycle through *endcycle* and then recurs. If the plug is in ($p = IN$) and the system is turned on ($t = ON$) and the captured temperature is less or equal than the preferred temperature ($tp \leq preferred$), it cools down the room by outputting the captured temperature (*cooldown*!tp), after which it indicates the end of the cycle through *endcycle* and then recurs. If either the plug is out ($p \neq IN$) or it is turned off (($t \neq ON$)), it simply performs *endcycle* before recursing.

The main action of the process is given after the symbol $\bullet$. It defines the process behaviour: the *CONTROLLER* performs the initialisation *INIT* and then (;) behaves like *CTR*. The reader can notice that there are two operators that indicate a sequence: the prefixing operator indicates a sequence between an occurrence of an event and an action, and the sequence operator (;) that indicates a sequence between two actions. These two operators also appear on the *SENSOR* process whose definition is shown as follows.

$$
\begin{array}{l}
\textbf{process}\ SENSOR\ \widehat{=}\ \textbf{begin} \\
\quad \textbf{state}\ SensorSt\ ==\ [memory : \mathbb{N}] \\
\quad INIT\ \widehat{=}\ memory := 0 \\
\quad SNSR\ \widehat{=}\ \mu X \bullet readtemp?nt \rightarrow memory := nt\ ;\ X \\
\qquad\qquad\qquad \Box\ gettemp!memory \rightarrow X \\
\qquad\qquad\qquad \Box\ switchoff\ \rightarrow Skip \\
\quad \bullet\ INIT\ ;\ SNSR \\
\textbf{end}
\end{array}
$$

The role of the *SENSOR* process is to capture the environment temperature and output it on channel *gettemp*. *SENSOR* has a state called *SensorSt* whose definition is the Z schema $[memory : \mathbb{N}]$. There are two actions on process *SENSOR*: *INIT* and *SNSR*. *INIT* initializes the memory with value 0 ($memory := 0$). On the other hand, *SNSR* has a fixed point X (μX) from which either ($\Box$): (1) event *readtemp*?*nt* is performed for capturing the temperature from the environment and then storing it on memory ($memory := nt$) and then (;) recursing to the fixed point of X, or (2) event *gettemp*!*memory* is performed for outputting the captured temperature on channel *gettemp* and then recursing to the fixed

point of *X*, or (3) switching off with *switchoff* and then skiping (*Skip*). The behaviour of *SENSOR*, delimited by •, is: execute action *INIT* and then (;) *SNSR*.

The *SENSOR* and the *CONTROLLER* processes compose process *AIRCONTROLLER*, that will be explained as follows.

$$\textbf{process}\ AIRCONTROLLER \mathrel{\widehat=} \\ (CONTROLLER \ [\![\{\!|\ gettemp, switchoff\ |\!\}]\!]\ SENSOR) \setminus \{\!|\ gettemp\ |\!\}$$

Process *AIRCONTROLLER* is a Parallel composition between processes *CONTROLLER* and *SENSOR*, synchronising on channels *gettemp* and *switchoff*. When an event is on the channel set of the Parallelism, it only occurs if both branches of the Parallelism are communicating that event. In the case of *AIRCONTROLLER*, there is a state where *gettemp*!*memory* (from *SENSOR*) synchronise on *gettemp*?*tp* (from *CONTROLLER*) making *tp* receive the value of variable *memory*. Channel *gettemp* is hidden from the external environment, meaning that inner communications on *gettemp* are not visible to the environment.

The Air Controller specification used a sub-set of operators of *Circus*, but it did not encompass all of them. Envisaging explaining all constructs of *Circus*, we will formalize its syntax and explain it. The syntax of *Circus* was firstly described on (Woodcock & Cavalcanti 2001), using Extended-BNF (EBNF) notation. It was then specified by (Freitas 2006, Freitas 2005), and, at last, by (Oliveira 2006*a*). We will also formalize an EBNF grammar for *Circus* based on the constructs quoted in (Oliveira 2006*a*), showing it production by production.

A *Circus* specification (CircusSpec syntactic category) is a sequence of zero or more paragraphs:

CircusSpec $\longrightarrow$ CircusPara*

Each paragraph of *Circus* can be a Z paragraph (ZedPara), a channel declaration, a channel set declaration or a process declaration (ProcDecl):

CircusPara $\longrightarrow$ ZedPara | **channel** ChannelDef | **chanset** N == CSExp | ProcDecl

The ZedPara syntactic category is defined on document (Spivey 1992) as the Paragraph syntactic category. This syntactic category allows the specification of paragraphs on the

Z-notation, like type definitions, axiomatic descriptions or Z-schemas. We give an example of an axiomatic description, extracted from (Spivey 1992), as follows:

$[\text{square}: \mathbb{N} \rightarrow \mathbb{N} \mid \forall\, n : \mathbb{N} \bullet \text{square}\ (n) = n * n]$

The above Z schema declares a total function, from natural number to natural number, called *square*, and whose definition is given by predicate: for all natural numbers n, $square(n)$ equals n times n.

The channel definition (ChannelDef) can be a simple channel definition or a composition of simple channel definitions. Each simple channel definition is formed by at least one channel name (N+) with a possible expression denoting the type of the channel (Expr). The type of the channel can be defined dinamically (in the moment a communication involving a channel occurs), and in this case the channel have generic parameters ([N+]). When one wishes to declare groups of channels, it can use schema expressions (SchemaExp):

ChannelDef $\longrightarrow$ SimpleChannelDef | SimpleChannelDef **;** ChannelDef
SimpleChannelDef $\longrightarrow$ N+ | N+ : Expr | [N+] N+ : Expr | SchemaExp

The channel set definition (ChannelDef) can contain a basic channel set expression ({| N* |}), a channel set call (N), an union between channel sets (CSExp $\cup$ CSExp), an intersection between channel sets (CSExp $\cap$ CSExp) or a set minus (CSExp $\setminus$ CSExp):

CSExp $\longrightarrow$ {| N* |} | N | CSExp $\cup$ CSExp | CSExp $\cap$ CSExp | CSExp $\setminus$ CSExp

The process declaration is formed by the reserved word **process** followed by the definition of the process (ProcDef) possibly preceded by generic parameters ([N+]):

ProcDecl $\longrightarrow$ **process** N = ProcDef | **process** N[N+] $\widehat{=}$ ProcDef

The definition of the process can be parameterised ($\bullet$), indexed ($\odot$) or normal. Parameterised processes are processes that have variables as parameters, indexed processes are processes that behave like themselves but having their channels renamed by indexed labels (for process P, if we instantiate an indexed call i : T $\odot$ P, then each channel c of P is renamed to c_i):

ProcDef $\longrightarrow$ Decl $\bullet$ ProcDef | Decl $\odot$ ProcDef | Proc

The Proc syntactic category represents the category for the definition of processes. They can be compound processes (for example: external choice, internal choice, parallelism, interleave, and sequential), unary processes (for example: rename and hide processes), iterated processes or call processes (possibly with parameters). We will show the derivations of Proc and explain them afterwards:

Proc $\longrightarrow$
begin Para* (**state** Para)? Para* $\bullet$ Action **end**
| Proc ; Proc | Proc $\Box$ Proc | Proc $\sqcap$ Proc | Proc |[CSExp]| Proc | Proc ||| Proc
| Proc $\setminus$ CSExp | (Decl $\bullet$ ProcDef)(Exp+) | N (Exp+) | N | (Decl $\odot$ ProcDef) $\lfloor$Exp+$\rfloor$
| N $\lfloor$Exp+$\rfloor$
| Proc [N+ := N+] | N [Exp+] | ; Decl $\bullet$ Proc | $\Box$ Decl $\bullet$ Proc | $\sqcap$ Decl $\bullet$ Proc
| |[CSExp]| Decl $\bullet$ Proc | ||| Decl $\bullet$ Proc

We explain each of these constructors as follows:

The **begin** Para* (**state** Para)? Para* $\bullet$ Action **end** syntactic category defines basic processes. This category is delimited by the reserved words **begin** and **end**. A basic process is formed by zero or more paragraphs, possibly a state declaration and a mandatory Main Action (denoted by $\bullet$ Action). The main action of a basic process defines the behavior of that process. For example: the process

begin
state [x : $\mathbb{N}$]
A $\widehat{=}$ c.0 $\rightarrow$ Skip
$\bullet$ A
end

has its behaviour delimited by the content of the action A, that synchronises on $c.0$ then skips successfully.

The Proc ; Proc syntactic category represents a sequence of two processes, in which the left process is performed before the right process.

The Proc $\Box$ Proc syntactic category represents an external choice between two processes, in which there is a deterministic offer of the first event of the left process and the first event of the right process to an external environment. Depending on the chosen

first event, this operator behaves like the left process (if the chosen event is from the left process) or like the right process (if the chosen event is from the right process). The Proc ⊓ Proc syntactic category represents an internal choice between two processes, in which the choice is performed non-deterministically, that is, the system performs the choice internally.

The Proc |[CSExp]| Proc syntactic category corresponds to an interface parallelism, in which both processes synchronise on the events of the CSPExpr channel set. When two processes synchronise on an event, it means that the event will occur only if both processes are simultaneously offering it. If this is not the case, the process deadlocks.

The Proc ||| Proc syntactic category corresponds to an interleaving between two processes. That means that they run simultaneously but without any synchronisation between events.

The Proc \ CSExp syntactic category represents a hiding of events of the CSExp channel set on Proc. When an event is hidden from the external environment, it cannot perform the event. For example:

begin • c → Skip **end** \ {| c |}

On the example above, *c* is hidden, so process **begin** • c → Skip **end** \ {| c |} behaves as Skip.

The (Decl • ProcDef)(Expr+) syntactic category defines a call of a parameterised process defined on the fly, which we will call "on the fly parameter process call". A process defined on-the-fly is a nameless process that can not be referenced. For example, the process

(x : ℕ • **begin** c.x → Skip **end**) (5)

defines an on-the-fly parameter process call that communicates the value 5 on channel *c* and then skips.

The N (Expr+) syntactic category defines a call process with parameters. In this case, there is at least one expression (Expr+) put as parameter. The N syntactic category corresponds to a call action, but with no parameters.

The (Decl ⊙ ProcDef) ⌊Exp+⌋ syntactic category defines a call of an indexed process defined on the fly. The indexed call renames each channel *c* with the name *c_i*, in which *i* is the index of the channel, and each channel communicates the value *i* on its first field. For example, the process (i : ℕ ⊙ c?x → Skip) ⌊ 5 ⌋ communicates *c*_5.5?*x*, receiving a

value on x, and then skips. The N $\lfloor$Exp+$\rfloor$ defines an indexed call process with parameters.

The Proc [N+ := N+] syntactic category defines pairs of channel renamings. Each name on the left side is the name of the channel that is being renamed, and the corresponding name on the right side is the new name. For example,

(**begin** • a $\rightarrow$ b $\rightarrow$ Skip **end**) [a, b := c, d]

is equivalent to

begin • c $\rightarrow$ d $\rightarrow$ Skip **end**

The N [Expr+] syntactic category defines a call to a process with generic type expressions that are used throughtout the process by channels with generic types.

The last five syntactic categories for processes define Iterated Operators on the form OP Decl • Proc, where OP $\in \{;, \Box, \sqcap, |||, [\![$ CSExp $]\!]$ Decl • Proc $\}$. Each Iterated Operator is a generalization of its corresponding operator. For example:

||| x : $\{1,2,3\}$ • **begin** • c.x $\rightarrow$ Skip **end**

is equivalent to

begin • c.1 $\rightarrow$ Skip ||| c.2 $\rightarrow$ Skip ||| c.3 $\rightarrow$ Skip **end**

The following grammar production exposes the syntax for *Circus* actions. A *Circus* action can be a CSP action, a Z schema or a command:

Action $\longrightarrow$ CSPAction | ZedSchema | Command

The syntactic category ZedSchema (defined on (Spivey 1992) as the syntactic category Schema-Text) defines *Circus* actions as Z Schemas. The syntactic category Pred is given by all First Order Logic predicates (Tannen 2009).

The productions for the syntactic category Command can be seen as follows:

Command $\longrightarrow$
N+ := Expr+ | **if** GActions **fi** | **var** Decl • Action

N+ : [Pred, Pred] | { Pred } | [Pred]
val Decl • Action | **res** Decl • Action | **vres** Decl • Action

The syntactic category N+ := Expr+ represents an Assignment command. It has one or more variable names on the left, and the same number of expressions on the right. For example:

x := 5 and x, y, z := 0, 1, 3

The syntactic category **var** Decl • Action represents a Variable Block command. It introduces one or more variables on the scope of an action. It has a declaration (Decl) with one or more variables and a type, and an action. For example:

x, y : $\mathbb{N}$ • x := 5

The syntactic categories **val** Decl • Action, **res** Decl • Action and **vres** Decl • Action are Substitution commands. They specify parameterised actions whose parameters are passed, respectively, by value (**val**), by result (**res**) and by value-result (**vres**).

The syntactic category N+ : [Pred, Pred] represents a Specification Statement. This construct was described on (Morgan & Gardiner 1990) and specifies parts of the program yet to be developed. The syntactic category N+ specifies a frame of variables (a frame is a set of variables that are considered, on the expression of the post-condition, for the satisfiability of the post-condition) and [Pred, Pred] is a tuple with a pre-condition (the first Pred) and a post-condition (the second Pred). The meaning is: if the pre-condition holds for the frame N+, then it is expected that the post-condition also holds, changing only values of the variables in the frame. Syntactic categories { Pred } and [Pred] are syntactic sugarings for special cases of a Specification Statement. It defines {pre} for [pre, true] (assumptions), and [post] for [true, post] (coercions).

The syntactic category **if** GActions **fi** defines an if-guarded command. This command has a set of branches that are guarded actions (GActions) and the command can behave like each action if their corresponding guards (predicates) are true. If there are more than one guard that is true, then it behaves like an internal choice between the branches whose guards are true. The syntactic category GActions is defined as follows:

GActions $\longrightarrow$ Pred $\rightarrow$ Action ([] Pred $\rightarrow$ Action)*. We define the syntactic categories

CSPAction, Comm and CParameter as follows:

CSPAction $\longrightarrow$
Skip | Stop | Chaos | Comm $\rightarrow$ Action | Pred & Action
| Action ; Action | Action $\Box$ Action | Action $\sqcap$ Action
| Action |[NSExp | CSExp | NSExp]| Action
| Action |||[NSExp | NSExp]||| Action
| Action $\setminus$ CSExp | Call (Exp+)? | μ N $\bullet$ Action
| ; Decl $\bullet$ Action
| $\Box$ Decl $\bullet$ Action | $\sqcap$ Decl $\bullet$ Action
| |[CSExp]| Decl $\bullet$ |[NSExp]| $\bullet$ Action
| ||| Decl $\bullet$ |||[NSExp]||| Action

Comm $\longrightarrow$ N CParameter* | N [Exp+] CParameter*
CParameter $\longrightarrow$?N | ?N : Pred | !Exp | .Exp

The actions Skip, Stop and Chaos are basic actions. Skip is a successfull terminating action, Stop is a deadlocked action, and Chaos is an inherently divergent action.

Comm $\rightarrow$ Action is a prefixing action. It establishes that the performance of Action depends on the occurrence of the event represented by communication Comm. For example:

event $\rightarrow$ ACT

means that action ACT is performed only if event occurs.

The syntactic category Comm is for communications. A communication is formed by a name (the name of the channel that performs the communication) and by zero or more fields, that can be: Input field (?), Output field (!) or Dot field (.). Each Input field can have a restriction (a Predicate) on the input values. For example: on the communication

$c?x$: (x $\in \{1, 2, 3\}$),

channel c accepts either 1, 2 or 3 as a communication value. When there is no restriction, the Input field accepts any value for x of the type of that field. For example: for channel c : $\mathbb{N}$, communication $c?x$ accepts any value that belongs to $\mathbb{N}$. Output and Dot fields have the same meaning: they only communicate a given expression. E.g: $c!0$ and $c.0$ communicate value 0 through channel c.

Pred & Action is a guarded action. When an action is guarded by a predicate Pred, it means that it is performed only if Pred is true. Otherwise it deadlocks.

The action operators of sequence (;), external and internal choice (□ and ⊓) behave as their correspondent operators for processes.

Action |[NSExp | CSExp | NSExp]| Action

means an interface parallelism with name sets. The behaviour is the same as its correspondent operator for processes, except for assignments whose variables do not belong to their correspondent name set. For example: (x, y := 0 |[x | { } | y]| x, y := 1) is equivalent to x := 0 |[{ } | { } | { }]| y := 1.Finally, Action |[NSExp | NSExp]| Action is equivalent to Action |[NSExp | { } | NSExp]| Action.

Action \ CSExp is the hiding operator for actions. It behaves exactly as its correspondent on processes: all channels that lie inside CSExp are hidden from the environment external to Action.

Call (Exp+)? is the syntactic category for call actions. Parameterised actions can be called with (possibly) expressions as parameters. For example, for parameterised action

ACT $\widehat{=}$ x : $\mathbb{N}$ • c.x → Skip,

there can be a call like ACT (0).

μ N • Action defines a recursive block. N is the identifier for the fixed point. For example:

μ X • a → X.

The syntactic categories □ Decl • Action, ⊓ Decl • Action, |[CSExp]| Decl • |[NSExp]| • Action, ||| Decl • |||[NSExp]|| Action are for replicated actions. They behave as their unfolded versions. For example:

⊓ x : { 1, 2, 3 } • c.x → Skip

behaves as

c.1 → Skip ⊓ c.2 → Skip ⊓ c.3 → Skip

Circus has a Denotational Semantics (Oliveira 2006*a*) and an Operational Semantics, that was firstly described in (Freitas 2006). They will be explained on the next sub-sections.

3.1.1 *Circus*' Denotational Semantics

The Denotational Semantics of *Circus* (Oliveira 2006*a*) is the theory of *Circus* on the Unifying Theories of Programming (UTP) (Hoare & Jifeng 1998). The UTP is a theory that allows unification of different semantics (Operational, Denotational and Algebraic): it represents programs specifying their states. The UTP has observational variables, that are variables that describe each state of a program. A program can have many observational variables, but there are four mandatory observational variables:

ok, ref, tr and $wait$.

Variable ok is a boolean that indicates if the program has started, variable ref is a set that stores the events that are refused, variable tr is a sequence that stores the trace (the sequence of events that were performed before), and $wait$ is a boolean variable that indicates if the program is in a waiting state or not.

All observational variables can be specified in terms of their initial states and their final states. The variable on its initial state is written as it is, and on its final state it has a dash (’). For example, if x means the value of x on its initial state, then x' means the value of x on its final state. For ok, ref, tr and $wait$, their final states are:

ok', ref', tr' and $wait'$.

Dashed variable ok' means successfull termination of the program, dashed variable ref' means the refused events on the final state, dashed variable tr' means the sequence of events that were performed on the final state, and dashed variable $wait'$ indicates if the program is in a final waiting state.

The theory of *Circus* on the UTP has predicate functions called Healthiness conditions: these functions establish conditions that must be satisfied by a given predicate in order for it to be considered a construct in *Circus*. *Circus* has three healthiness conditions for *Circus* actions: R1, R2 and R3. They are going to be explained as follows.

$$R1(P) \mathrel{\widehat{=}} P \wedge tr \leq tr' \tag{3.1}$$

The healthiness condition $R1$ says that the history of events cannot be undone. This is established by the expression $tr \leq tr'$. The trace in the final state cannot be less than the trace in the initial state. Now we will present healthiness condition R2:

$$R2(P(tr, tr')) \mathrel{\widehat{=}} P(<>, tr' - tr) = P[<>, tr' - tr/tr, tr'] \tag{3.2}$$

R2 says that the behaviour of the process independs from what hapenned before. The expression above replaces, on the predicate P, tr by the empty trace ($<>$) and tr' by tr' - tr. Thus, the history of events on tr is irrelevant for the behaviour of the process. Before explaining R3, we will introduce the formula of a construct called Reactive Skip (II_{rea}) as follows:

$$II_{rea} \mathrel{\widehat{=}} (\neg ok \wedge tr' \leq tr) \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v) \tag{3.3}$$

The expression above stands for: The process either did not start (having its history of events not undone) or has finished (ok') having its UTP variables unchanged. Thus the behaviour of R3 is: if the process is waiting for other processes to finish, then it either did not start or has already finished. Otherwise it continues. Now we will show and explain R3:

$$R3(P) \mathrel{\widehat{=}} II_{rea} \lhd wait \rhd P \tag{3.4}$$

R3 says that processes that wait for other processes to finish must not start. This expression on formula 3.1.1 says: if *wait* is true, then the conditional expression equals the reactive skip (II_{rea}). Otherwise it equals P.

We define, as follows, the function R, that is a composition of the three healthiness conditions already defined:

$$R(P) = R1 \circ R2 \circ R3(P) \tag{3.5}$$

Function R (P) is a composition of healthiness conditions R1, R2 and R3. It is used to define the meaning of each *Circus* construct on the UTP. Another operator that is used by all *Circus* constructs is the Design. The meaning of the design is:

$$Pre \vdash Post \mathrel{\widehat{=}} Pre \wedge ok \Rightarrow Post \wedge ok' \tag{3.6}$$

The expression above means: if the program has started in a state in which its pre-conditions holds, then when it finishes its post-condition holds.

The Design construct is used on the denotational definition of each *Circus* action on

the UTP. Thus, the following formula holds:

$$Circus\ Action = R(Pre \vdash Post) \tag{3.7}$$

Formula 3.7 indicates that every *Circus* action is a reactive design $R(Pre \vdash Post)$. The full list of healthiness conditions for all constructs of *Circus* (including *Circus* processes) can be found at (Oliveira 2006*a*). The definitions of the reactive designs for the constructs of *Circus* can be seen at (Oliveira 2006*a*). Some of them lie on annex B.

We will show the definition of Woodcock's Operational Semantics on the following sub-section.

3.1.2 Woodcock's Operational Semantics

Cavalcanti and Gaudel (Cavalcanti & Gaudel 2011, J. C. P. Woodcock 2007) described an updated Operational Semantics for *Circus*. It shows different kinds of definitions for both labelled and silent transitions, in which the nodes contain a constraint, that indicates if that node is enabled or not. On this paper, the node is represented by a triple $(c \mid s \models A)$, where c is the constraint, s is the sequence of assignments and A is the action that remains to be executed. A similar kind of description is shown on Woodcock's technical report (Cavalcanti & Woodcock 2013), where the Operational Semantics of a language that has similar constructs to *Circus* (CML) is shown and explained.

Before defining Woodcock's Operational Semantics, we define here what is a loose constant. Loose constants are symbolic constants declared throughout the rules of the Operational Semantics. They are named w possibly with an index. The denotational definition of each rule of this Semantics has loose constants and is presented in the sequel:

Definition 3.1.
$(c_1 \mid s_1 \models A_1) \xrightarrow{\tau} (c_2 \mid s_2 \models A_2)$
$= \forall\ w \,.\, c_1 \wedge c_2 \Rightarrow Lift\ (s_1)\ ;\ A_1 \sqsubseteq Lift\ (s_2)\ ;\ A_2$

Definition 3.2.
$(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)$
$= \forall\ w \,.\, c_1 \wedge c_2 \Rightarrow Lift\ (s_1)\ ;\ A_1 \sqsubseteq (Lift\ (s_2)\ ;\ c.w1 \rightarrow A_2) \,\Box\, (Lift\ (s_1)\ ;\ A_1)$

where Lift (s) = R1 ∘ R3 (true ⊢ s ∧ tr' = tr ∧ ¬ wait')

Definition 3.1 gives the denotational meaning of a Silent Transition. It means: for all loose constants w, if c_1 and c_2 are true, then the left side of the transition is refined by the

right side of the transition. The meaning of the denotational definition on 3.2 is: if c_1 and c_2 are true, then the left side of the transition is refined by the external choice between the right side of the transition prefixed by the label and the left side of the transition.

Transition Rules Work (Cavalcanti & Woodcock 2013) also gives a definition for timed constructs and defines rules for timed constructs and designs. Our interest, however, lies on the semantics of the constructs that are common with *Circus*. The rules are shown on the following format:

$$\frac{P}{Transition}$$

where P is a condition/assumption of existence for the transition. It is possible that the condition P is formed by more than one factor, and in this case, the condition can be seen as the conjunction (and) between the factors (for example: rules for Assignment 1, Prefixing Input 2, Prefixing Output 3, Guard 11, and etc) The technical report (Cavalcanti & Woodcock 2013) defines 26 rules for constructs that lie on *Circus*. These constructs are: Assignment, Prefixing, Variable Block, Sequence, Internal Choice, External Choice, Parallelism, Hiding and Recursion. The rules for these constructs will be shown and explained below.

Rule 1.

Assignment:

$$\frac{c \qquad (s\ ;\ (w0 = e))}{(c \mid s \models v := e) \xrightarrow{\tau} (c \wedge (s\ ;\ (w0 = e)) \mid s\ ;\ v := w0 \models Skip)}$$

Rule 1 establishes that if constraint c (factor c from the assumption), from the source node, is true and loose constant $w0$ equals expression e after assignment s (factor s; $(w0 = e)$ from the assumption), then there is a silent transition that goes from $(c \mid s \models v := e)$ to $(c \wedge (s;\ (w0 = e)) \mid s;\ v := w0 \models \text{Skip})$. The constraint c is updated to

$$c \wedge (s\ ;\ (w0 = e)),$$

the state s is updated to $s;\ v := w0$, and the program text v := e goes to Skip. For example:

$$(\text{true} \mid \{\} \models \text{x} := 0) \xrightarrow{\tau} (\text{true} \wedge (\text{w0} = 0) \mid \{\text{x} := \text{w0}\} \models \text{Skip})$$

Rule 2.

***Prefixing Input**:*

$$\frac{c \quad T \neq \emptyset \quad x \notin \alpha(s)}{(c \mid s \models d?x{:}T \rightarrow A) \xrightarrow{d.w0} (c \wedge w0 \in T \mid s \,;\, \textbf{var}\ x := w0 \models \textbf{let}\ x \bullet A)}$$

Rule 2 establishes that if type T is non-empty and input variable x does not lie on the alphabet of assignment s, then prefixing actions with an input communication will have a labelled transition consuming event $d.w0$ and reaching a final node with the constraint $c \wedge w0 \in T$. The factor $w0 \in T$ is a result of the occurrence of the event of the labelled transition. The assignment is sequenced by $x := w0$ and the program text goes to **let** x • A. The **let** construct does not lie on the syntax of *Circus*, being only a flag to indicate the insertion of variable x on the scope of action A. We give as follows an example of transition involving an input prefixing:

$$(\mathrm{c} \mid \mathrm{s} \models \mathrm{d?x{:}}\mathbb{N} \rightarrow \mathrm{Skip}) \xrightarrow{d.w0} (\mathrm{c} \wedge \mathrm{w0} \in \mathbb{N} \mid \mathrm{s} \,;\, \textbf{var}\ \mathrm{x} := \mathrm{w0} \models \textbf{let}\ \mathrm{x} \bullet \mathrm{Skip})$$

Rule 3.

***Prefixing Output**:*

$$\frac{c \quad s \,;\, w0 = e}{(c \mid s \models d!e \rightarrow A) \xrightarrow{d.w0} (c \wedge s \,;\, (w0 = e) \mid s \models A)}$$

Rule 3 establishes that if constraint c is true and loose constant $w0$ equals expression e then there will be a labelled transition (with label $d.w0$) going from (c | s $\models$ d!e $\rightarrow$ A) to (c $\wedge$ s ; (w0 = e) | s $\models$ A), in which the factor s; $(w0 = e)$ is a consequence of the occurrence of event $d.w0$ and program text evolves from d!e $\rightarrow$ A to A itself. For example:

$$(\mathrm{c} \mid \mathrm{s} \models \mathrm{c!0} \rightarrow \mathrm{A}) \xrightarrow{c.w0} (\mathrm{c} \wedge \mathrm{s} \,;\, (\mathrm{w0} = 0) \mid \mathrm{s} \models \mathrm{A})$$

The Variable Block command (syntactic category **var** Decl • Action) has three rules: Variable Block Begin, Variable Block Visible and Variable Block End.

Rule 4.

Variable Block Begin:

$$(c \mid s \models \textbf{\textit{var}}\ x : T \bullet A) \xrightarrow{\tau} (c \wedge w0 \in T \mid s\ ;\ \textbf{\textit{var}}\ x := w0 \models \textbf{\textit{let}}\ x \bullet A)$$

Rule 4 simply includes variable x on the scope of action A (going from program text **var** $x : T \bullet A$ to program text **let** $x \bullet A$). Constraint c from the source node is strengthened by adding the factor w0 ∈ T and the assignment is sequenced by **var** x := w0.

Rule 5.

Variable Block Visible:

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \textbf{\textit{let}}\ x \bullet A_1\ ) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{\textit{let}}\ x \bullet A_2)}$$

Rule 5 advances the program text from **let** x • A_1 to **let** x • A_2, provided that (c_1 | s_1 ⊨ A_1) $\xrightarrow{l}$ (c_2 | s_2 ⊨ A_2), no matter if the arc is labelled or silent. For example:

(true | {} ⊨ **let** x • x := 0) $\xrightarrow{\tau}$ (true ∧ (w0 = 0) | {x := w0} ⊨ **let** x • Skip)

provided that

(true | {} ⊨ x := 0) $\xrightarrow{\tau}$ (true ∧ (w0 = 0) | {x := w0} ⊨ Skip)

Rule 6.

Variable Block End:

$$\frac{c}{(c \mid s \models \textbf{\textit{let}}\ x \bullet Skip) \xrightarrow{\tau} (c \mid s\ ;\ \textbf{\textit{end}}\ x \models Skip)}$$

Rule 6 ends the scope of x if the program text is Skip.

Rule 7.

***Sequence Progress**:*

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1 \; ; B) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \; ; B)}$$

Rule 7 establishes the progress of a sequence action. If we have the sequence A_1 ; B and we know that there is a transition that progresses from A_1 to A_2, then we will have a transition for the sequence operator with similar configurations, but that progresses from A_1 ; B to A_2 ; B. For example:

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip} \; ; \text{b} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip} \; ; \text{b} \rightarrow \text{Skip})$$

provided that

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip})$$

Rule 8.

***Sequence End**:*

$$\frac{c}{(c \mid s \models \mathit{Skip} \; ; A) \xrightarrow{\tau} (c \mid s \models A)}$$

Rule 8 just ends the sequence through a silent transition. It happens when the left side of the sequence is Skip.

Rule 9.

Internal Choice Left*:

$$(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\tau} (c \mid s \models A_1)$$

Rule 10.

***Internal Choice Right**:*

$(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\tau} (c \mid s \models A_2)$

Rules 9 and 10 are similar. Rule 9 makes a silent transition between a node whose program text is an internal choice ($A_1 \sqcap A_2$) and a node that has the left side of the internal choice as program text, and 10 makes a silent transition between a node whose program text is an internal choice ($A_1 \sqcap A_2$) and a node that has the right side of the internal choice as program text. For example: The set of transitions for action $a \rightarrow Skip \sqcap b \rightarrow Skip$ is:

$(true \mid \{\} \models a \rightarrow Skip \sqcap b \rightarrow Skip) \xrightarrow{\tau} (true \mid \{\} \models a \rightarrow Skip)$ [1]
$(true \mid \{\} \models a \rightarrow Skip \sqcap b \rightarrow Skip) \xrightarrow{\tau} (true \mid \{\} \models b \rightarrow Skip)$ [2]

Rule 11.

***Guard**:*

$$\frac{c \qquad (s \,;\, g)}{(c \mid s \models g \,\&\, A) \xrightarrow{\tau} (c \wedge (s \,;\, g) \mid s \models A)}$$

Rule for guarded actions exists if constraint c is true and if guard g is true after occurrence of assignment s from the left side of the transition. Then it establishes that the transition consumes, on the program text, guard g and strengthens constraint c with $s \,;\, g$. For example:

$(true \mid \{\} \models x = 0 \,\&\, Skip) \xrightarrow{\tau} (x = 0 \wedge (\{\} \,;\, true) \mid \{\} \models Skip)$

$\widehat{=}$

$(true \mid \{\} \models x = 0 \,\&\, Skip) \xrightarrow{\tau} (true \wedge x = 0 \mid \{\} \models Skip)$

Rule 12.

External Choice Begin*:*
$(c \mid s \models A_1 \Box A_2) \xrightarrow{\tau} (c \mid s \models (\textbf{loc}\ s \bullet A_1)\ [+]\ (\textbf{loc}\ s \bullet A_2))$

Rule 12 makes the program text advance to a state where local copies from the states (**loc** s) of both sides of the external choice are stored. The operator [+] is called Extra Choice, and is purely a syntactic device to establish that a parallel execution is taking place.

Rule 13.

External Choice Skip Left*:*
$(c \mid s \models loc\ s_1 \bullet Skip\ [+]\ loc\ s_2 \bullet A) \xrightarrow{\tau} (c \mid s \models Skip)$

Rule 14.

External Choice Skip Right*:*
$(c \mid s \models loc\ s_1 \bullet A\ [+]\ loc\ s_2 \bullet Skip) \xrightarrow{\tau} (c \mid s_2 \models Skip)$

Rules 13 and 14 are similar. They terminate an external choice by silently leading the program text to Skip, if one of the branches of the external choice is Skip.

Rule 15.

External Choice End*:*

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3)}{(c \mid s \models loc\ s_1 \bullet A_1\ [+]\ loc\ s_2 \bullet A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3)}$$

Rule 15 ends the external choice by checking the existent labelled arcs going out from the branches. They will be the labelled arcs that go out from the node whose program text is the external choice. For example:

$$(\text{true} \mid \{\} \models \text{loc}\ \{\} \bullet \text{a} \rightarrow \text{Skip}\ [+]\ \text{loc}\ \{\} \bullet \text{b} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip})$$
$$(\text{true} \mid \{\} \models \text{loc}\ \{\} \bullet \text{a} \rightarrow \text{Skip}\ [+]\ \text{loc}\ \{\} \bullet \text{b} \rightarrow \text{Skip}) \xrightarrow{b} (\text{true} \mid \{\} \models \text{Skip})$$

provided that

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip})$$
$$(\text{true} \mid \{\} \models \text{b} \rightarrow \text{Skip}) \xrightarrow{b} (\text{true} \mid \{\} \models \text{Skip})$$

Rule 16.

External Choice Silent Left*:*

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\tau} (c_3 \mid s_3 \models A_3)}{(c \mid s \models loc\ s_1 \bullet A_1\ [+]\ loc\ s_2 \bullet A_2) \xrightarrow{\tau} (c \mid s \models loc\ s_3 \bullet A_3\ [+]\ loc\ s_2 \bullet A_2)}$$

Rule 16 does not resolve the external choice. Instead of it, it silently advance the left branch, that produces the silent transition, and makes the program reach the external choice between the advanced branch and the other branch.

Rule 17.

***External Choice Silent Right**:*

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{\tau} (c_3 \mid s_3 \models A_3)}{(c \mid s \models loc\ s_1 \bullet A_1\ [+]\ loc\ s_2 \bullet A_2) \xrightarrow{\tau} (c \mid s \models loc\ s_1 \bullet A_1\ [+]\ loc\ s_3 \bullet A_3)}$$

Rules 16 and 17 does not resolve the external choice. Instead of it, they silently advance the branch that produces the silent transition and makes the program reach the external choice between the advanced branch and the other branch. For example:

(true | {} $\models$ **let** x $\bullet$ Skip $\Box$ a $\rightarrow$ Skip) $\xrightarrow{\tau}$ (true | {} ; **end** x $\models$ Skip $\Box$ a $\rightarrow$ Skip)

provided that

(true | {} $\models$ **let** x $\bullet$ Skip) $\xrightarrow{\tau}$ (true | {} ; **end** x $\models$ Skip)

Rule 18.

Parallel Begin*:

$(c \mid s \models A_1 \ [\![s_1 \mid cs \mid s_2]\!]\ A_2)$
$\xrightarrow{\tau}$
$(c \mid s \models (loc\ (s \mid x_1)_{+x2} \bullet A_1) \ [\![x_1 \mid cs \mid x_2]\!]\ (loc\ (s \mid x_2)_{+x1} \bullet A_2))$

The first rule for Parallelism 18 makes a transition from the parallelism to a state where the branches are partitioned according to their name sets. (s | x_1) restricts the sequence of assignments to those whose variables on the left sides lie on the set x_1. The expression (s | x_1)$_{+x2}$ restricts the set of names on the assignment to x_1 and allows assignments whose left variable names lie on $x2$. For example:

$(x, y, z := 0, 1, 2) \mid \{x\} = (x := 0)$.

Local copies (loc) from both partitioned assignments are stored on each parallel branch.

Rule 19.

Parallel End*:

$(c \mid s \models (loc\ s_1 \bullet Skip) \ [\![x1 \mid cs \mid x2]\!]\ (loc\ s_2 \bullet Skip)) \xrightarrow{\tau} (c \mid (s_1 \mid x1) \wedge (s_2 \mid x2) \models Skip)$

Rule 19 ends Parallelism when Skip is reached on both branches. Furthermore, the assignments s are updated to (s_1 | x1) $\wedge$ (s_2 | x2).

Rule 20.

Parallel Independent Left (PIL):

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \qquad (l = \tau) \vee chan(l) \notin cs}{(c \mid s \models loc\ s_1 \bullet A_1 \ [\![x1 \mid CS \mid x2]\!]\ loc\ s_2 \bullet A_2) \xrightarrow{\tau} (c \mid s \models loc\ s_3 \bullet A_3 \ [\![x1 \mid CS \mid x2]\!]\ loc\ s_2 \bullet A_2)}$$

Rule 21.

Parallel Independent Right (PIR):

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3 \qquad (l = \tau) \vee chan(l) \notin cs}{(c \mid s \models loc\ s_1 \bullet A_1 \ [\![x1 \mid CS \mid x2]\!]\ loc\ s_2 \bullet A_2) \xrightarrow{\tau} (c \mid s \models loc\ s_1 \bullet A_1 \ [\![x1 \mid CS \mid x2]\!]\ loc\ s_3 \bullet A_3)}$$

Rules 20 and 21 describe transitions that occur in a parallel composition when l is silent or the channel of the label l is not on the channel set. In these transitions, only one branch advances. In the case of Parallel Independent Left (PIL), only the left branch advances. In the case of Parallel Independent Right (PIR), only the right branch advances. For example:

$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip} \ |||\ \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip} \ |||\ \text{a} \rightarrow \text{Skip})$

and

$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip} \ |||\ \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip} \ |||\ \text{Skip})$

provided that

$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip})$ and the channel set of the interleave operator ($|||$) is empty.

Rule 22.

***Parallel Synchronised**:*

$$
\frac{\begin{array}{l} d \in cs\ c_1\ c_2\ c_3\ c_4\ (w_1 = w_2) \\ (^*,\diamond) \in (?, !), (!, ?), (!, !), (., .), (?, ?) \\ (c_1 \mid s_1 \models A_1) \xrightarrow{d^* w_1} (c_3 \mid s_3 \models A_3) \\ (c_2 \mid s_2 \models A_2) \xrightarrow{d \diamond w_2} (c_4 \mid s_4 \models A_4) \end{array}}{\begin{array}{l} (c_1 \wedge c_2 \mid s \models (c_1 \mid loc\ s_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid loc\ s_2 \bullet A_2)) \\ \xrightarrow{d|w_2} \\ \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \models \\ \left(\begin{array}{l} (c_3 \wedge (w_1 = w_2) \mid locs_3 \bullet A_3) \\ [\![x_1 \mid cs \mid x_2]\!] \\ (c_4 \wedge (w_1 = w_2) \mid locs_4 \bullet A_4) \end{array} \right) \end{array} \right) \end{array}}
$$

Rule 22 specifies the advance of the parallelism when both branches have labelled transitions with the same channel and this channel lies on the channel set of the parallelism. In this case, both branches advance. For that to happen, all constraints from both initial and destination nodes from each branch must be true and the decoration of each communication must be a combination of the decorations on the set $\{?, !, .\}$. For example:

$$
\begin{array}{l} (\text{true} \mid \{\} \models \text{a?x} \rightarrow \text{Skip} \,[\![\, \{\!|\, \text{a} \,|\!\}\,]\!]\, \text{a.0} \rightarrow \text{b} \rightarrow \text{Skip}) \\ \xrightarrow{a.w0} \\ (\text{true} \wedge \text{w0} \in \mathbb{N} \wedge \text{w1} = 0 \wedge \text{w0} = \text{w1} \mid \{\} \models \text{Skip} \,[\![\, \{\!|\, \text{a} \,|\!\}\,]\!]\, \text{b} \rightarrow \text{Skip}) \end{array}
$$

provided that

$$
\begin{array}{l} (\text{true} \mid \{\} \models \text{a?x} \rightarrow \text{Skip}) \xrightarrow{a.w0} (\text{true} \wedge \text{w0} \in \mathbb{N} \mid \{\} \models \text{Skip}) \\ \wedge\ (\text{true} \mid \{\}\ \text{a.0} \rightarrow \text{b} \rightarrow \text{Skip}) \xrightarrow{a.w1} (\text{true} \wedge \text{w1} = 0 \mid \{\} \models \text{b} \rightarrow \text{Skip}) \\ \wedge\ \text{a} \in \{\!|\, \text{a} \,|\!\} \end{array}
$$

Rule 23.

Hiding Internal:

$$\frac{(c_1 \mid s_1 \models A) \xrightarrow{l} (c_1 \mid s_1 \models B) \quad l \in S}{(c_1 \mid s_1 \models A \setminus S) \xrightarrow{\tau} (c_1 \mid s_1 \models B \setminus S)}$$

Rule 23 specifies a silent advance on a transition for a hide action (A \ S) when, on the rule of the hidden action (which is A), there is a labelled advance with an arc ($\xrightarrow{l}$) whose channel lies on the set of hiding (l ∈ S). In this case, as the channel of the arc is hidden, it is converted to a silent advance (through a τ arc). For example:

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip} \setminus \{\mid \text{a} \mid\}) \xrightarrow{\tau} (\text{true} \mid \{\} \models \text{Skip} \setminus \{\mid \text{a} \mid\})$$

provided that

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip}) \\ \wedge\ \text{a} \in \{\mid \text{a} \mid\}$$

Rule 24.

Hiding Visible:

$$\frac{(c_1 \mid s_1 \models A) \xrightarrow{l} (c_1 \mid s_1 \models B) \quad l \notin S}{(c_1 \mid s_1 \models A \setminus S) \xrightarrow{l} (c_1 \mid s_1 \models B \setminus S)}$$

Rule 24 specifies a labelled advance when the channel of the arc does not lie on the hiding set. In this case, the hide action advances through a similar arc of the provided condition:

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip} \setminus \{\mid \text{b} \mid\}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip} \setminus \{\mid \text{b} \mid\})$$

provided that

$$(\text{true} \mid \{\} \models \text{a} \rightarrow \text{Skip}) \xrightarrow{a} (\text{true} \mid \{\} \models \text{Skip}) \\ \wedge\ \text{a} \notin \{\mid \text{b} \mid\}$$

Rule 25.

Recursion:

$$\frac{(c \mid s \models A) \xrightarrow{\tau} (c \mid s \models B)\ N = A}{(c \mid s \models N) \xrightarrow{\tau} (c \mid s \models B)}$$

Rule 25 specifies rule for a recursive call. The meaning is: if there is a silent transition from (c | s ⊨ A) to (c | s ⊨ B) and A is the body (see definition C.4) of an action whose name is N (N = A), then there is also a silent transition from (c | s ⊨ N) to (c | s ⊨ B). We exemplify the application of the rule above to a *Circus* process:

process P $\widehat{=}$ **begin**
A $\widehat{=}$ a → Skip
• A
end

In the case above, we can see that there is an action (A) whose definition is a → Skip. The rule for this action is:

(true | {} ⊨ A) $\xrightarrow{\tau}$ (true | {} ⊨ a → Skip)

On the main action (•), we can see that there is a call to action A (satisfying the condition N = A). In this case, the rule is the same above.

The rules defined so far are suitable for only a sub-set of *Circus* actions. Nevertheless, as we envisage a full Operational Semantics for *Circus* in order to develop an LPTS Compiler for *Circus*, we will extend Woodcock's Semantics to more *Circus* actions and also for *Circus* processes.

3.1.3 Extending Woodcock's Operational Semantics to *Circus*

One of the contributions we gave for Woodcock's Operational Semantics is to extend its set of rules. We extended Woodcock's rules for Silent and Labelled Transitions, adapted two rules from (Cavalcanti & Gaudel 2011) to Woodcock's definitions of Silent and Labelled Transitions and lifted Woodcock's semantics to the level of *Circus* processes.

New Rules for Silent and Labelled Transitions

Silent and labelled transitions are used for both *Circus* actions and basic processes. One of the contributions of this thesis was the creation of additional rules for (J. C. P. Woodcock 2007, Cavalcanti & Gaudel 2011, Cavalcanti & Woodcock 2013), envisaging the implementation of an LPTS Compiler for *Circus* based on these rules. So, the sequence of rules from rule 26 to rule 37 are contributions of this PhD thesis using Silent and Labelled Transitions. In what follows, we will show these rules.

Rule 26.

***Hiding Skip**:*
$(c \mid s \models Skip \setminus S) \xrightarrow{\tau} (c \mid s \models Skip)$

Rule 26 is a rule for consuming the channel set of the hiding action. When the program text is formed only by the Skip action, there are no channels to be hidden, so the hide set can be removed through a silent transition.

Rule 27.

***If-Guarded Command**:*
Be

$IGC \mathrel{\widehat{=}} \textbf{if } (pred_1) \rightarrow A_1 \,[\!]\, (pred_2) \rightarrow A_2 \,[\!]\, \ldots \,[\!]\, (pred_n) \rightarrow A_n \textbf{ fi}$, *then*

$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_1 \mid s \models A_1)$
$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_2 \mid s \models A_2)$
...
$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_n \mid s \models A_n)$
$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge (\neg\ pred_1) \wedge \ldots \wedge (\neg\ pred_{n-1}) \wedge (\neg\ pred_n) \mid s \models Chaos)$

Rule 28 defines a set of rules for If-Guarded Commands. Each rule shows a possibility of computation for the program text, each one depending on a predicate from the If-Guarded Command. There is a possibility in which the command goes to the node whose program text is A_1 and whose constraint is $c \wedge pred_1$ (indicating that A_1 is reachable only if $pred_1$ is true), and so on. When all guards are false, the If-Guarded Command diverges (with Chaos), and when more than one guard is true, the If-Guarded Command behaves as an internal choice between the true-guarded branches.

Rule 28.

***Call Action**:*

$$\frac{CALL = Content\ (CALL)}{(c \mid s \models CALL) \xrightarrow{\tau} (c \mid s \models Content\ (CALL))}$$

The semantics of a call action is formed by the transitions of the body (content) of the action. As in the LPTS we have to consume the action call, we will make a transition that brings the node of the call to the node of the call's body. CALL is the name of the call action, and <content> is its content.

Rule 29.

***Call Action With Parameters**:*

$$\frac{CALL = Content\ (CALL)}{(c \mid s \models CALL\ (v_1, v_2, ..., v_n)) \xrightarrow{\tau} (c \mid s \models \textbf{var}\ p_1, p_2, ..., p_n : T \bullet p_1, p_2, ...\ p_n := v_1, v_2, ..., v_n\ ;\ <content>)}$$

When there are parameters in the call action, we have to do some transformations into the action to make a correct expansion of the content of the action. We have to declare the parameters into the scope of the action and assign the call values to these parameters. In 29, v_i are the call values of each parameter and p_i are the variable names of each parameter on the definition of the action.

Before explaining rule 30, we will define a function called *Ren*. This function has an action and a sequence of tuples in which each tuple is a renaming pair. For example:

Ren (a $\rightarrow$ b $\rightarrow$ Skip, [(a, c), (b, d)]) = c $\rightarrow$ d $\rightarrow$ Skip.

Rule 30.

***Rename Action**:*

$$(c \mid s \models A\ [v1, v2, ..., vn := t1, t2, ..., tn\ ]) \xrightarrow{\tau} (c \mid s \models Ren\ (A, [(v1, t1), (v2, t2), ...\ (vn, tn))])$$

In which $Ren(A, < Seq.of\ Tuples >)$ is the renamed version of the action. In order to consume the renaming, we only have to replace the channel names according to the renaming list. When there is nested renaming, we only have to aggregate pairs replacing the right sides accordingly after resolving the outermost renaming. For example:

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow Skip\ [a, d := b, f]\ [b, g, i := c, h, j] =$
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow Skip\ [a, d, g, i := c, f, h, j]$

Rule 31.

***Alphabetised Parallel Action**:*
$(c \mid s \models A_1 \ [\![NS1 \mid CS1 \parallel CS2 \mid NS2]\!]\ A_2)$
$\xrightarrow{\tau}$
$(c \mid s \models (A_1 \ [\![\Sigma - CS1]\!]\ Stop)\ [\![NS1 \mid CS1 \cap CS2 \mid NS2]\!]\ A_2\ [\![\Sigma - CS2]\!]\ Stop)$

The Operational Semantics of CML gives us 5 rules for parallelism, but only for interface parallelism (the parallelism in which there is only a channel set and the events synchronise in this channel set). We formalized on 31 the rule for alphabetised parallelism using a transition that transforms the alphabetised parallelism into an interface parallelism. The rule above works for non-terminating processes. As the events in the alphabetised parallelism synchronise only in the channel set formed by the intersection of each alphabet, there can be a rule (shown in 31) to make this transformation. Σ is the set of all events.

The proof of this rule is left as a piece of future work.

Rule 32.

***Parameter Action Call**:*
$(c \mid s \models (p_1 : T_1, p_2 : T_2, ..., p_n : T_n \bullet A)\ (expr_1, expr_2, ..., expr_n))$
$\xrightarrow{\tau}$
$(c \mid s \models Repl\ (A, (expr_1, expr_2, ..., expr_n)))$

This rule corresponds to the syntactic category (ParamAction)(Expr+) in the *Circus* grammar. It is different from the rule 3.54 because it is a call action whose content is defined on the fly. All we have to do to consume this action is to replace the call expressions of (Expr+) into the occurrences of the parameter variables on the ParamAction. $Repl(Action, (expr_1, ..., expr_n))$ is the action with the parameters replaced by the expressions of ($expr_1$, ..., $expr_n$). For example:

$(true \mid \{\} \models (x, y, z : \mathbb{N} \bullet c!(x + y + z) \rightarrow Skip)\ (3, 4, 5))$
$\xrightarrow{\tau} (true \mid \{\} \models c!(3 + 4 + 5) \rightarrow Skip)$

Rule 33.

***Iterated Actions**:*
Let $T = \{elem_0, elem_1, ..., elem_n\}$;
Let $x_i, y_i, z_i, ...$ *be a variable for iteration;*
Let $OP \in \{ \Box, \sqcap, ;, \|[CS]\|, ||| \}$, *then*

$(c \mid s \models ITOP\ x_0, x_1, ..., x_n : T \bullet A\ (x_0, x_1, ..., x_n))$
$\xrightarrow{\tau}$
$(c \mid s \models IteratedExpansion\ (A, Decl, ITOPFLAG))$

Rule 33 shows a rule for any iterated operator *ITOP*. The rule unfolds the iterated operator into its correspondent compound operator (through IteratedExpansion, which is defined on C.3). *IteratedExpansion* receives three parameters: (1) the action *A*; (2) the declaration Decl of variables used to iterate, and (3) a flag *ITOPFLAG* that indicates what operator is being iterated. For example:

IteratedExpansion (a.x $\rightarrow$ *Skip*, x : $\{0, 1, 2, 3\}$, EXTCHOICE) =
a.0 $\rightarrow$ *Skip* $\Box$ a.1 $\rightarrow$ *Skip* $\Box$ a.2 $\rightarrow$ *Skip* $\Box$ a.3 $\rightarrow$ *Skip*

On the above example, EXTCHOICE is a flag that indicates that the operator that is being iterated is an external choice ($\Box$).

Rule 34.

***Parameter Action**:*
$(c \mid s \models p_1 : T_1, p_2 : T_2, ..., p_n : T_n \bullet A) \xrightarrow{\tau} (c \wedge p_1 \in T_1 \wedge p_2 \in T_2 \wedge ... p_n \in T_n \mid s \models A)$

Rule 35.

***Specification Statements**:*
$(c \mid s \models V : [Pre, Post]) \xrightarrow{\tau} (c \wedge Pre \mid s \models V\ [:]\ [Pre, Post])$
$(c \mid s \models V : [Pre, Post]) \xrightarrow{\tau} (c \wedge (\neg\ Pre) \mid s \models Skip)$
$(c \wedge Pre \mid s \models V\ [:]\ [Pre, Post])$
$\xrightarrow{\tau}$
$(c \wedge Pre \wedge Post \wedge (\alpha(P) - V)' = (\alpha(P) - V)) \mid s \models Skip)$

Specification statements establish that if the precondition is true, then we have to assure that the state of the program that comes next satisfies the postcondition over the

variables on the frame of the statement.

The rules for Specification statement, thus, would be very simple: There would be an initial node from which two transitions would go out: one in which the precondition is true, and one where it is false. From the destination node where the precondition is true, there would be a change in the program text with the addition of a syntactic device (like the extra choice for external choice), and the constraint would have an and factor with the precondition. We call the syntactic device ”extra statement” ([:]) C.31.

Adapted rules from Cavalcanti and Gaudel

Between the rules defined by Cavalcanti and Gaudel (Cavalcanti & Gaudel 2011), one of them was for basic processes and the other one for Z Schemas. We adapted these rules using Woodcock's definitions for Silent and Labelled transitions. We show their definitions on this sub-section:

Rule 36.

***Z Schema**:*

$$\frac{c \wedge (s \,;\, \textbf{\textit{pre}}\ Op)}{(c \mid s \models Op) \xrightarrow{\tau} (c \wedge (s \,;\, Op\ [w/v']) \mid s;\ v := w \models Skip)}$$

The rule for Z Schema defines the operational behaviour for any Z construct of *Circus*. The operational behaviour of any Z Schema **Op** is to silently (τ) advance to Skip changing its constraint by adding the factor $(s;\ Op[w/v'])$: this factor indicates that after sequence s of assignments, the predicate $Op[w/v']$ must be true (substitution $[w/v']$ replaces each occurrence of a dashed variable by a loose constant w). This silent advancement only happens if predicate

$$c \wedge (s;\ \textbf{pre}\, Op)$$

holds. Factor $(s;\ \textbf{pre}\, Op)$ indicates that, after sequence s of assignments, the pre-condition **pre** of **Op** must be true. The **pre** operator is called pre-condition investigation, and returns the pre-condition of predicate **Op**. Its formal definition lies on C.2.

All rules we described so far are applied to *Circus* actions. These rules use Silent (τ) or Labelled transitions to describe advances of the program. Nevertheless, *Circus* has two level of constructs: the level of *Circus* actions and the level of *Circus* processes. The most elementar *Circus* process is the basic process, that contain a state and a set of action paragraphs and whose behaviour is defined by a main action ($\bullet$). We describe rule Basic Process Begin as follows:

Rule 37.

***Basic Process Begin**:*

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{ls} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \textbf{begin state } [\textit{Vars-decl} \mid \textit{inv}]\ \textit{PARAS} \bullet A_1 \textbf{ end}) \xrightarrow{ls} (c_2 \mid s_2 \models \textbf{begin state } [\textit{Vars-decl} \mid \textit{inv}]\ \textit{PARAS} \bullet A_2 \textbf{ end})}$$

The rule 37 makes the connection between the level of *Circus* processes and the level of *Circus* actions, by saying that: if there is a rule that links *Circus* action A_1 with *Circus* action A_2 by an arc *ls* (that can be Labelled or Silent), then there is also a transition that links *Circus* process

begin state [Vars-decl | inv] PARAS $\bullet$ A_1 **end** ,

to *Circus* process

begin state [Vars-decl | inv] PARAS $\bullet$ A_2 **end**

with the same arc.

Lifting the Operational Semantics to *Circus* processes

Circus has two level of constructs: the level of *Circus* processes and the level of *Circus* actions. We, alongside (J. C. P. Woodcock 2007, Cavalcanti & Gaudel 2011, Cavalcanti & Woodcock 2013), have defined so far rules for *Circus* actions and *Circus* basic processes, for which each node has the following configuration:

$(c \mid s \models \mathit{ActOrBasicProc})$

where *ActOrBasicProc* is a program text that is a *Circus* action or a basic process. The rules for configuration above used Labelled and Silent (τ) Transitions. Rules for compound and other non-basic processes could be determined by manually refining the process into a basic process using the refinement laws of (Oliveira 2006*a*). Nevertheless, as our intention is to verify a tool (***JCircus***) whose acceptable grammar (shown at annex A) encompasses compound and non-basic processes, we need an LPTS Compiler that also encompasses these processes. Thus, we need an Operational Semantics that also formalises the process of transformation of each compound and non-basic process into a basic process. The configuration (c | s $\models$ Program) has a sequence of assignments *s* that compose the advancement of the main action of the *Circus* basic process: as the main action advances, the sequence of state assignments also may advance. For non-basic processes, however, the sequences of state assignments are encapsulated on the process branches and do not advance. So, we overload the configuration of each node of the LPTS by creating nodes whose program texts are non-basic processes (called *NonBasicProc* below), as follows:

$$(c \models \mathit{NonBasicProc})$$

The syntactic transition is exclusively applied to *Circus* processes and allows the transformation of each *Circus* process to a Basic Process, inside which the rules for *Circus* actions and rule "Basic Process Begin" (37) can be applied. Thus, another contribution of this thesis is that we lifted the Operational Semantics on (J. C. P. Woodcock 2007, Cavalcanti & Gaudel 2011, Cavalcanti & Woodcock 2013) to *Circus* processes by creating another kind of transition, the Syntactic Transition ($\rightsquigarrow$). A syntactic transition mandatorily have a source node with configuration $(c \models \mathit{NonBasicProc})$, but the destination node can be both a $(c \models \mathit{NonBasicProc})$ or a $(c \mid s \models \mathit{BasicProc})$. So, the transition

$$(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2)$$

is valid, and the transition

$$(c_1 \models P_1) \rightsquigarrow (c_2 \mid s \models \mathit{BasicProc})$$

is also valid. The denotational definition of Syntactic Transition is shown as follows:

Definition 3.3. *Syntactic Transition:*
$(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2) =$
$\forall\, w \,.\, c_1 \wedge c_2 \Rightarrow ((Lift\ (getAssignments\ (P_1))\ ;\ P_1 \sqsubseteq Lift\ (getAssignments\ (P_2))\ ;\ P_2))$
$\wedge\ Lift\ (getAssignments\ (P_1)) = Lift\ (getAssignments\ (P_2))$

$(c_1 \models P_1) \rightsquigarrow (c_2 \mid s \models BasicProc) =$
$\forall\, w \,.\, c_1 \wedge c_2 \Rightarrow ((Lift\ (getAssignments\ (P_1))\ ;\ P_1 \sqsubseteq Lift\ (s)\ ;\ BasicProc))$
$\wedge\ Lift\ (getAssignments\ (P_1)) = Lift\ (s)$

getAssignments is an auxiliary function that calculates the sequence of assignments of a node whose program text is a process. The definition of *getAssignments* is shown on C.1. A node whose program text is a process is defined by a constraint and a process, represented as (c $\models$ P). The transition ($c_1 \models P_1$) $\rightsquigarrow$ ($c_2 \models P_2$) means that, if c_1 and c_2 are true, then P_1 can be syntactically transformed to P_2 without semantically changing the program. The Syntactic Transition, thus, does not specify a path of computation on the program. The idea is to syntactically transform the process in order to reach a Basic Process. During this transformation, inner assignments on the process cannot change (Lift (getAssignments (P_1)) = Lift (getAssignments (P_2))) and the assignments have to preserve healthiness. The rules can be defined as follows.

Rule 38.

***Basic Process Reduction**:*
Based on C.2.16 and C.2.17 from the refinement calculus of Circus:

$$\frac{c \qquad \alpha(STA) \cap \alpha(STB) = \emptyset}{(c \models (\textbf{begin state } STA\ PARS\text{-}A \bullet A\ \textbf{end})\ OP\ (\textbf{begin state } STB\ PARS\text{-}B \bullet B\ \textbf{end})) \rightsquigarrow (c \mid s \models \textbf{begin state } (STA \wedge STB)\ (PARS\text{-}A \wedge_{\Xi} STB)\ (PARS\text{-}B \wedge_{\Xi} STA) \bullet A\ OP\ B\ \textbf{end})}$$

Where $OP \in \{ \sqcap, \Box, ;, |||, [\![CS]\!], [\![\alpha(A) \mid CS \mid \alpha(B)]\!] \}$

Rule Basic Process Reduction (38) reduces a compound process between basic processes into a single basic process with a compound main action (through theorems C.2.16 and C.2.17), provided that constraint c is true and each basic process state has disjunct alphabets ($\alpha(STA) \cap \alpha(STB) = \emptyset$). The resulting basic process has state $(STA \wedge STB)$, and its set of paragraphs is formed by the union of two subsets of paragraphs:

(PARS-A $\wedge_{\Xi}$ STB),

and

(PARS-B $\wedge_{\Xi}$ STA).

The set of paragraphs (PARS-A $\wedge_{\Xi}$ STB) means that in each paragraph, no state component from STB on PARS-A will be changed, and this is indicated by operator "$\wedge_{\Xi}$". The set (PARS-B $\wedge_{\Xi}$ STA) is analogous to the previous one, but now keeping state components of STA unchanged on PARS-B. The main action (•) of the resulting basic process is A OP B, where OP is the binary (compound) operator. The Basic Process Reduction rule can be applied to any binary operator (OP $\in \{ \sqcap, \Box, ;, |||, [\![\text{CS}]\!], [\![\alpha(\text{A}) \mid \text{CS} \mid \alpha(\text{B})]\!] \}$). For example:

(true $\models$ (**begin state** [x:$\mathbb{N}$] • a→Skip **end**) $\Box$ (**begin state** [y:$\mathbb{N}$] • b→Skip **end**))
$\rightsquigarrow$
(true | {} $\models$ **begin state** [x : $\mathbb{N} \wedge$ y : $\mathbb{N}$] • a $\rightarrow$ Skip $\Box$ b $\rightarrow$ Skip **end**)

Rule 39.

Compound Process Left:

$$\frac{(c_1 \models P1) \rightsquigarrow (c_2 \models P3)}{(c_1 \models P1\ OP\ P2) \rightsquigarrow (c_2 \models P3\ OP\ P2)}$$

Where $OP \in \{ \sqcap, \Box, ;, |||, [\![CS]\!], \ ; \ \}$

Rule 40.

Compound Process Right (for $OP \in \{ \Box, \sqcap, [\![CS]\!], |||, \}$***):***

$$\frac{(c_1 \models P1) \rightsquigarrow (c_2 \models P3) \qquad OP \in \{ \Box, \sqcap, [\![CS]\!], ||| \}}{(c_1 \models P1\ OP\ P2) \rightsquigarrow (c_2 \models P1\ OP\ P3)}$$

Rules 39 and 40 implement an advancement of a branch of a compound process. It can happen on the left branch or on the right branch. The meaning is: if a process branch can be syntactically advanced to another process, then the compound process that contains that branch can also be syntactically transformed advancing the branch that can be advanced. For example:

Be

P $\widehat{=}$ (true $\models$ (**begin state** [x: $\mathbb{N}$] $\bullet$ a $\rightarrow$ Skip **end**) $\Box$ (**begin state** [y: $\mathbb{N}$] $\bullet$ b $\rightarrow$ Skip **end**)),
Q $\widehat{=}$ (**begin state** [y: $\mathbb{N}$] $\bullet$ c $\rightarrow$ Skip **end**), and
R $\widehat{=}$ (true $\models$ **begin state** [x : $\mathbb{N} \wedge$ y : $\mathbb{N}$] $\bullet$ a $\rightarrow$ Skip $\Box$ b $\rightarrow$ Skip **end**)

(true $\models$ P $|||$ Q) $\rightsquigarrow$ (true $\models$ R $|||$ Q)

provided that

(true $\models$ P) $\rightsquigarrow$ (true $\models$ R)

Rule 41.

Hiding Advance:

$$\frac{(c_1 \models P1) \rightsquigarrow (c_2 \models P2)}{(c_1 \models P1 \setminus S) \rightsquigarrow (c_2 \models P2 \setminus S)}$$

If P1 can syntactically be transformed to P2, then P1 $\setminus$ S can be syntactically transformed to P2 $\setminus$ S too.

Rule 42.

Hiding Basic Process:

$(c \models (\textbf{begin state } ST\ PARS \bullet A \textbf{ end}) \setminus S) \rightsquigarrow (c \mid s \models \textbf{begin state } ST\ PARS \bullet (A \setminus S) \textbf{ end})$

If the Hide Process hides a Basic Process with channel set S, then it can be syntactically converted to a Basic Process whose main action is hidden by S.

Rule 43.

Rename Advance:

$$\frac{(c_1 \models P1) \rightsquigarrow (c_2 \models P2)}{(c_1 \models P1\ [a1, a2, ... = b1, b2, ...]) \rightsquigarrow (c_2 \models P2\ [a1, a2, ... = b1, b2, ...])}$$

If P1 can syntactically be transformed to P2, then P1 [a1, a2, ... = b1, b2, ...] can be syntactically transformed to P2 [a1, a2, ... = b1, b2, ...] too.

Rule 44.

Rename Basic Process:

$(c \models (\textbf{begin state } ST\ PARS \bullet A \textbf{ end})\ [a1, a2, ... = b1, b2, ...])$

$\rightsquigarrow$

$(c \mid s \models \textbf{begin state } ST\ PARS \bullet (A\ [a1, a2, ... = b1, b2, ...]) \textbf{ end})$

If the Rename Process renames a Basic Process with channel assignment pairs [a1, a2, ... = b1, b2, ...], then it can be syntactically converted to a Basic Process whose main action is renamed by [a1, a2, ... = b1, b2, ...].

Rule 45.

Parameterless Call Process:

$(c \models P) \rightsquigarrow (c \models Content\ (P))$

Rule 46.

Call Process with normal parameters:

$(c \models P\ (N+)) \rightsquigarrow (c \models ParamContent\ (P, [N+]))$

Rule 47.

Call Process with indexed parameters:

$(c \models P \lfloor N+ \rfloor) \rightsquigarrow (c \models IndexedContent\ (P, [N+]))$

Call processes (either parameterless or having parameters, indexed or not) can be syntactically transformed to their contents. The definitions of Content, ParamContent and IndexedContent can be seen at C.4, C.6, C.5.

Rule 48.

***Iterated Processes**:*
(c ⊨ ITOP Decl • P) ⇝ (c ⊨ IteratedExpansion (P, Decl, ITOPFLAG))
IteratedExpansion is defined on C.3.

Iterated Operators can be transformed to their expanded versions. No infinite types are allowed to these operators (both for Iterated Actions and Iterated Processes). The definition of IteratedExpansion is on C.3. As we saw on rule 33, *IteratedExpansion* receives three parameters: (1) the action *A*; (2) the declaration Decl of variables used to iterate, and (3) a flag *ITOPFLAG* that indicates what operator is being iterated.

Chapter 4

JCircus

This chapter aims at explaining the translator from *Circus* to Java. Before detailing the translator, we will expose some basic concepts that will be useful to explain the translator.

4.1 JCSP

Java Communicating Sequential Processes (JCSP) (Welch 2000, Welch et al. 2010) is a Java API that provides constructs that allow the implementation of some CSP primitives in Java. JCSP allows the implementation of CSP channels and processes. Each process, in JCSP, is a class that implements the `CSProcess` interface. The behaviour of the process in JCSP is given by the definition of its `run` method.

Differently from processes, that are uniquely implemented as classes that implement the `CSProcess` interface, channels in JCSP can be implemented using different constructs depending on their use: if the channel communicates one value and is used only in a single-synchronised environment (for example: a process that communicates $a?x$ interacts with a process that communicates $a!0$), then it may be implemented as an attribute, for example, of class `One2OneChannelSymmetric`. Nevertheless, if the channel is used on a multi-synchronised environment, then it may be implemented using the `AltingBarrier` construct.

Let's suppose that we have the following CSP process:

P = $c!0 \rightarrow SKIP$

We can implement, in JCSP, process P as follows:

```
import org.jcsp.lang.*;
public class P implements CSProcess {
        One2OneChannelSymmetric c;
        public P (One2OneChannelSymmetric c) {
                this.c = c;
        }
        public void run() {
                c.out().write (0);
                (new Skip ()).run();
        }
}
```

P is a CSP process, thus, in JCSP, it is a class that implements the `CSProcess` interface. It has a `One2OneChannelSymmetric` attribute (called `c`), that is the implementation of channel c of process P. P can be instantiated using a constructor that stores the initialised channel `c`. The behaviour $c!0 \rightarrow SKIP$ is implemented inside method `run`. $c!0$ is implemented as instruction `c.write (0)` and *SKIP* is implemented as `(new Skip ()).run()`. The prefixing operator $\rightarrow$ can be implemented as the Java instruction separator "`;`". Now suppose CSP process Q as follows:

Q = $c?x \rightarrow SKIP$

We can implement, in JCSP, process Q as follows:

```
import org.jcsp.lang.*;
public class Q implements CSProcess {
        One2OneChannelSymmetric c;
        public Q (One2OneChannelSymmetric c) {
                this.c = c;
        }
        public void run() {
                Object x = c.in().read ();
                (new Skip ()).run();
        }
}
```


The implementation of process Q is formed by a constructor that stores an input front-end

in `c` and a `run` method that implements behaviour $c?x \rightarrow SKIP$, where $c?x$ is implemented as `Object x = c.read ()`. The other constructs are similar to those on process P.

We consider now an implementation of a process R, a parallel composition between P and Q:

R = P |[{| c |}]| Q

The JCSP code of process R is shown as follows:

```
public class R implements CSProcess {
        One2OneChannelSymmetric c;
        public R (One2OneChannelSymmetric c) {
                this.c = c;
        }
        public void run() {
                P p = new P (c);
                Q q = new Q (c);
                Parallel par = new Parallel (new CSProcess [] {p, q});
                par.run();
        }
}
```

Object `c`, on the implementation of process R, is a `One2OneChannelSymmetric` with an input (?) and an output (!) front-end. The output front-end is passed on constructor `c`, and the input is passed on constructor of process R. Then `p` and `q` are put in parallel on variable `par`, that is run with `par.run()`. Parallelism between processes that synchronise on the intersection of their events can be implemented by the `Parallel` operator. There is also a main method that allows execution of process R from the Java Virtual Machine. Figure 4.1 shows what happens with channel `c` during execution of process R.

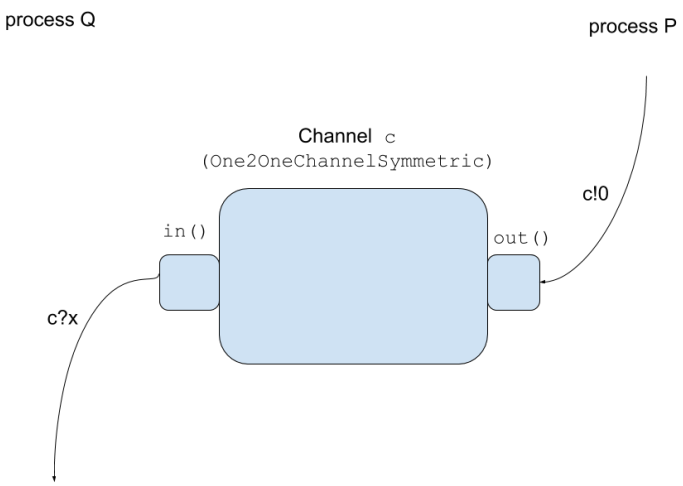

Figure 4.1: Channel c (One2OneChannelSymmetric) during execution of process R

On figure 4.1, processes P and Q are running in parallel. Process P synchronises on front-end `out()` and communicates value 0. Value 0, then, is passed for front-end `in()` and stored on variable `x`. P remains in a deadlocked state until Q communicates value 0. Channel `One2OneChannelSymmetric` mandatorily demands an input and an output communication in order to avoid a deadlock.

Process R is a parallel composition between two other processes that engage on a synchronisation on channel c. There is a situation, however, where there are more than two synchronisation engagements of processes: in this situation, it is said that there is a multi-synchronisation. In JCSP, if the channel does not communicate values and is used at least once in a multi-synchronised environment, then it can be implemented using `AltingBarrier`. The `AltingBarrier` construct allows the implementation of a barrier, that is a construct that has a set of front-ends. When a JCSP program reaches a barrier, it only can go forward if all front-ends were fulfilled by one or more JCSP processes. That matches the behaviour of Multi-synchronisation. `AltingBarrier` is also feasible for implementing channels that do not communicate values. Consider the following example:

$P = P_1 = P_2 = P_3 = a \rightarrow SKIP$
$PPP = P_1 \ [|\ \{|\ a\ |\}\ |]\ P_2 \ [|\ \{|\ a\ |\}\ |]\ P_3$
We show on figure 4.2 a barrier representation of process PPP in which P_1, P_2 and P_3 synchronise on channel a.

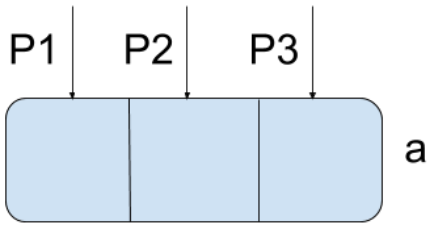

Figure 4.2: Figure showing an example of Multi-synchronisation

On figure 4.2, we can see that there are three front-ends on barrier *a* (that represent the implementation of channel *a*), in which P_1, P_2 and P_3 synchronise and such that each front-end receives only one of these processes. The synchronisation is made when the front-end of each `CSProcess` invokes `sync`. If at least one front-end do not receive a synchornisation, the parallel composition remains in a deadlocked state. In JCSP, processes P_1, P_2 and P_3 have a similar implementation, except for their names. We will show the implementation of process P (similar to P_1, P_2 and P_3) as follows:

```
public class P implements CSProcess {
     AltingBarrier a;
     public P (AltingBarrier a) {
          this.a = a;
     }
     public void run () {
          a.sync();
     }
}
```

Process P has an `AltingBarrier` as constructor and its behaviour is delimited by `a.sync()`. We show the implementation of process PPP (the one of figure 4.2) as follows:

```
public class PPP implements CSProcess {
     private AltingBarrier [] b;
     public PPP (AltingBarrier [] b) {
          this.b = b;
     }
     public void run () {
          P P1 = new PPP (b [0]);
          P P2 = new PPP (b [1]);
          P P3 = new PPP (b [2]);
```

```
            (new Parallel (new CSProcess [] {P1, P2, P3})).run();
            /**Line*/
        }
}
```

The implementation of process PPP has an array of `AltingBarrier` as a constructor: each element of this array is a front-end. Its `run` method instantiates objects `P1`, `P2` and `P3`, that are CSP processes of type `P`, in which each one receives a different front-end as parameter of constructor. The execution only reaches `/**Line*/` when all three processes have invoked their `sync` methods.

External choice can be implemented using the `Alternative` construct. This construct is initialized with a set of front-ends (`Guard`) and put these front-ends in alternation (that is, in a state where one of them can be accepted). That matches the behaviour of an external choice between a set of prefixing events. The chosen front-end is determined by the method `select`, that returns an index corresponding to the chosen front-end. We will define process S and give its implementation in JCSP:

S = a $\rightarrow$ SKIP $\Box$ b $\rightarrow$ STOP

Process S performs an external choice between branches $a \rightarrow SKIP$ and $b \rightarrow STOP$. We will show the implementation of process S as follows:

```
import org.jcsp.lang.*;
public class S implements CSProcess {
        AltingBarrier a, b;
        public S (AltingBarrier a, AltingBarrier b) {
                this.a = a; this.b = b;
        }
        public void run() {
                int s = (new Alternative (new Guard [] {a, b})).select();
                switch (s) {
                        case 0: {(new Skip ()).run(); break;}
                        case 1: {(new Stop ()).run(); break;}
                }
        }
}
```

As channels *a* and *b* from process S do not communicate values, they are implemented as alting barriers. The constructor of S receives two alting barriers, `a` and `b` and store them on the attributes of S, that have the same name. The external choice (□) between a and b is implemented as the instruction

```
int s = (new Alternative (new Guard [] {a, b})).select();
```

On the instruction above, `a` and `b` are guards: all instances of `AltingBarrier` extend `Guard`. The array of `Guard` is a parameter of `Alternative`, indicating that `a` and `b` are in alternance. Method `select()` will return the index of the guard that was selected from the external environment: it will be stored on integer variable `s`, and, depending on the chosen guard, it will behave as Skip (`(new Skip ()).run()`) or Stop (`(new Stop ()).run()`).

A system that runs in parallel with process P_1 (that is $a \rightarrow SKIP$) and process S performs event `a` and then skips, because the choice is made by process P_1. Figure 4.3 illustrates this situation.

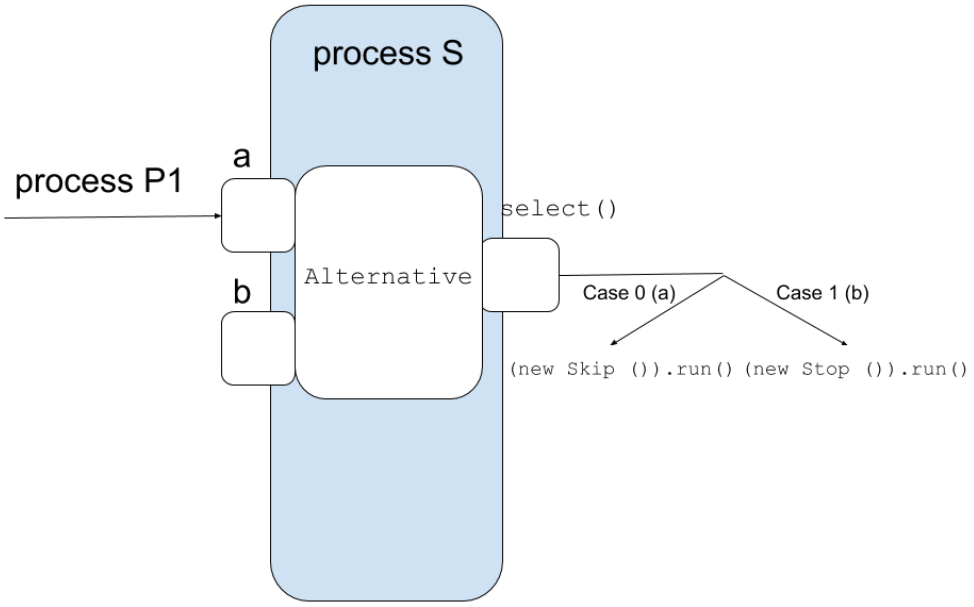

Figure 4.3: Processes P_1 and S running in parallel, interacting with each other

On figure 4.3, we can see that there is an interaction, in parallel, between process P_1 and process S. Process S offers a choice between events a and b and its behaviour afterwards depends on the choice made by the external environment (in this case, process P_1).

Limitations that JCSP has considerably increases the complexity of the implementation of CSP programs. The main limitations of JCSP are:

- Multi-synchronisation without communication;
- Communication only in single-synchronised environments and mandatorily between an input front-end and an output front-end of a channel;
- `Parallel` only supports interleaving between events of different names, not of events with similar names;
- Alternation only between front-ends, not between sets of front-ends or processes;

More examples of JCSP can be found at (Barrocas 2011). On the next section, we will explain the basic concepts about Community Z Tools.

4.2 Community Z Tools (CZT)

The Community Z Tools (CZT) (Malik & Utting 2005) is an initiative that provides a set of tools for dealing with Z-based languages, like Object-Z and *Circus*. CZT has two tools for *Circus*: a parser and a typechecker. The parser of *Circus* accepts a ".tex" file as input and generates an Abstract Syntax Tree with the attributes from the specification.

On the AST, all classes are subclasses of `Term`. A specification is an object of type `Spec`. A `Spec` has a list of sections (`Sect`), and each section can have a list of paragraphs (superclass `Para`). `ChannelPara`, `ChannelSetPara`, `AxPara`, `FreePara` and `ProcessPara` extend `Para`. `ProcessPara` has an attribute `ZName` and a `CircusProcess`. Figure 4.4 shows a diagram that represents the inheritance hierarchy for `CircusProcess`.

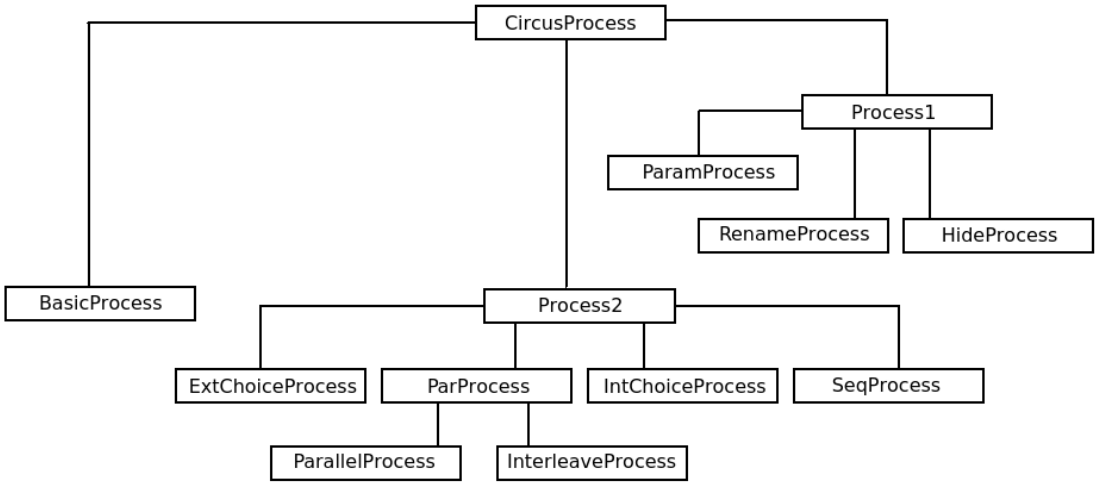

Figure 4.4: Inheritance Hierarchy for CircusProcess

4.3 LaTeX *Circus*

LaTeX *Circus* is the syntax for writing parseable specifications in *Circus* by the CZT parser of *Circus*. LaTeX *Circus* has the following major environments:

- circus: represented by \begin{circus} ... \end{circus}. The definition of every *Circus* paragraph (process paragraphs, channel paragraphs and channel set paragraphs) must be inside a *Circus* environment. They have to be divided into one or more environments (depending on the preference of the user);
- circusaction: represented by \begin{circusaction} ... \end{circusaction}. It must be used for actions when a process is written in LaTeX in a multi-environment manner;
- zed: represented by \begin{zed} ... \end{zed}. Used for declaring constants and defining types;
- axdef: represented by \begin{axdef} ... \end{axdef}. Used for declaring axiomatic definitions;
- schema: represented by \begin{schema} ... \end{schema}. Used for declaring Zed schemas;

The specification

FT ::= Elem1 | Elem2
channel a : FT
channel b : FT
process P $\widehat{=}$ **begin**
• a?x $\rightarrow$ SKIP $\Box$ b.Elem2 $\rightarrow$ SKIP
end

can be written in LaTeX as

```
\begin{zed}
FT ::= Elem1 | Elem2 \\
\end{zed}

\begin{circus}
\circchannel a : FT \\
\circchannel b : FT \\
\circprocess P \circdef \circbegin \\
     \circspot a?x \then \Skip \extchoice b.Elem2 \then \Skip \\
```

```
\circend \\
\end{circus}
```

The specification was written using a single *Circus* environment. Nevertheless, if we had to write it with multiple *Circus* environments we could write:

```
\begin{zed}
FT ::= Elem1 | Elem2 \\
\end{zed}

\begin{circus}
\circchannel a : FT \\
\circchannel b : FT \\
\end{circus}

\begin{circus}
\circprocess P \circdef \circbegin \\
     \circspot a?x \then \Skip \extchoice b.Elem2 \then \Skip \\
\circend
\end{circus}
```

The above LaTeX writing (with two *Circus* environments) would also be correct. Word **channel** is written as `\circchannel`, word **process** is written as `\circprocess`, symbol $\widehat{=}$ is written as `\circdef`, word **begin** is written as `\circbegin`, operator $\Box$ is written as `\extchoice`, and the prefixing operator $\rightarrow$ is written as `\then`. The `\\` is mandatory for paragraphs that lie on an environment that includes more than one paragraph. The syntax of LaTeX for *Circus* constructs can be seen at (Freitas 2008).

4.4 *JCircus*

JCircus (Freitas 2005, Barrocas 2011, Barrocas & Oliveira 2012) is a code generator that translates concrete specifications written in *Circus* into Java code. ***JCircus*** gets as input a ".tex" file written in LaTeX *Circus* and generates a code that uses the JCSP API. The tool was originally developed in 2005 by (Freitas 2005) and implemented a translation strategy described in (Freitas 2005). Then it was extended in 2011 by (Barrocas 2011), when it was provided support for Multi-Synchronisation, Complex Communications and General Interleaving, among other minor extensions. The tool was also integrated with CRefine (Oliveira et al. 2008) at that occasion.

JCircus has 8 modules. They are:

- The *parser*, implemented by the Community Z Tools (CZT) (Malik & Utting 2005), generates the Abstract Syntax Tree (AST) of a given *Circus* specification with attributes that can be accessed by other modules;
- The *typechecker*, also implemented by the CZT, type checks the specification;
- The *refiner* replaces calls to actions by the body of each action and refines iterated and compound processes to basic processes whose main actions have the operator of the iterated/compound process. This step is necessary for the parallelism updater and front-end updater steps;
- The *parallelism updater* implements a strategy to enforce the interleaving of channels (Barrocas & Oliveira 2012), which is needed because JCSP does not enforce the interleaving of channels that have the same name;
- The *input field variable updater* renames input field variables adding a label to each name;
- The *front-end updater* annotates each communication with an environment that maps each channel to its front-end set;
- The *pre-processor* collects information about channels, variables and processes from the specification and checks if the specification is translatable or not;
- the *translator* translates the specification into JCSP.

4.4.1 Modes of Execution

JCircus has three modes of execution:

- Via CRefine: on this mode, the user can load the CRefine tool, open one specification and translate it to Java. Figure 4.5 shows the screen of CRefine;
- Via ***JCircus***' Graphical User Interface: on this mode (4.6), the user can load a ".tex" specification and specify the path and the name of the folder in which it desires to generate the code;
- Via Benchmarking Session: on this mode (4.7), ***JCircus*** can be executed to more than one specification. There is a text file called *benchs.txt*. On this file, it is accepted firstly a number that specifies the number of specifications that will be translated, and then there are the names of the specifications with a boolean value on the right that indicates if the user desires to generate GUI for interacting with the generated code;

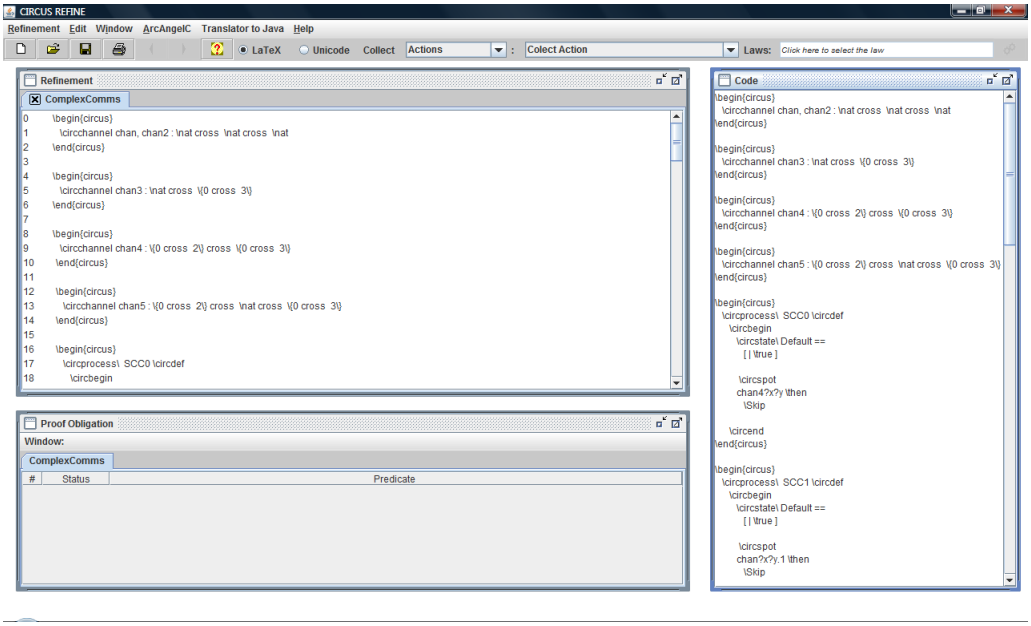

Figure 4.5: CRefine screen, one of the modes for executing ***JCircus***

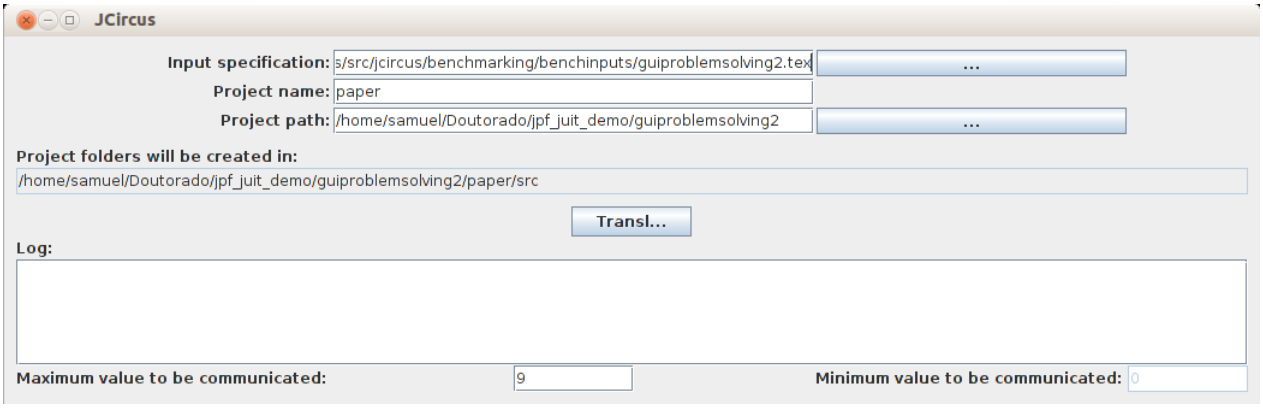

Figure 4.6: ***JCircus*** screen, another mode for executing ***JCircus***

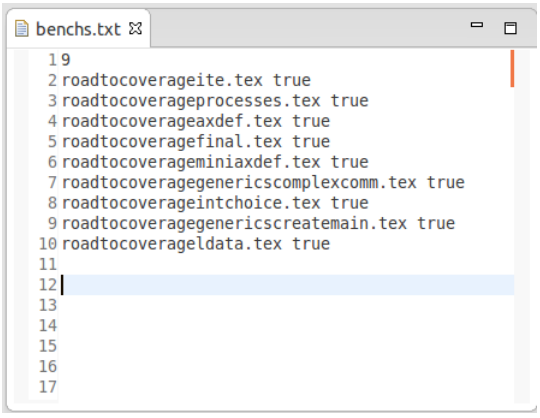

Figure 4.7: Benchs file, that is read by the Benchmarking Session

JCircus has an extremely simple GUI in which the user simply selects the input specification file and defines the name of the output java project. Finally, the user is asked to choose the system's main process. This choice defines the behaviour of the program that is automatically generated, if the specification is compliant with ***JCircus*** requirements. If, however, the specification is not compliant with the tool's requirements, or in case of any compilation errors or type errors, the translation is aborted and errors messages are displayed.

4.4.2 The requirements of the tool

JCircus has some requirements that must be satisfied in order to allow the translation from *Circus* to Java. Firstly, the specification must be concrete. If one desires to translate an abstract specification in *Circus*, it must refine the specification into a concrete and translatable *Circus* specification. Moreover, not all the constructs of *Circus* are translatable. The only translatable paragraphs of Z are those of free type declarations and axiomatic definitions, and even those must have specific forms to be translated. We will detail all those requirements below.

1. A *Circus* program is well formed and well typed (the parser and the typechecker assure this requirement);

2. The only types supported by ***JCircus*** are free types and $\mathbb{A}$ (an abstract type from the type system of *Circus* that encompasses integer and natural types). The $\mathbb{A}$ type is encoded in LaTeX-*Circus* as the word `\nat`. Although, in Unicode, `\nat` represents natural numbers ($\mathbb{N}$), ***JCircus*** interprets it as the $\mathbb{A}$ type and requires a maximal absolute number as input;

3. Z paragraphs have specific forms of declaration and use. They can be either axiomatic definition paragraphs (in this case, they are declared on the form $v : T \mid v = e$), free type paragraphs, or schemas of the form $[x_{11}, ..., x_{1n} : T_1; \ ...; \ x_{n1}, ..., x_{nn} : T_n \mid pred]$ that are used exclusively to define the state of a basic process (inside a process paragraph);

4. Hiding of channels can only be made through Hide Processes (Hide Actions are not translatable by ***JCircus*** at all), and even these may have restrictions. First of all, (1) the Hide Process can not have nested hiding, and (2) the channels that can be hidden are only those where Hiding can be ignored. The cases where Hiding can be ignored are (A) when the channel set CS of hiding does not contain any channel that is used

by the process (Channels (P) $\cap$ CS = $\emptyset$) and (B) when there is inner (single or multi) synchronisation on the hidden channels (like on (c?x $\rightarrow$ Skip $[\![\{\!|$ c $|\!\}]\!]$ c.0 $\rightarrow$ Skip $\setminus$ $\{\!|$ c $|\!\}$). We show these restrictions on the following examples:

channel $cs, cs2$
process $A \mathrel{\widehat=}$ **begin** $\bullet$ $cs \rightarrow Skip$ **end**
process $B \mathrel{\widehat=} A \setminus \{\!|\ cs2\ |\!\}$
process $C \mathrel{\widehat=} A \setminus \{\!|\ cs\ |\!\}$

On the example above, $Channels(B) = \{\!|\ cs\ |\!\}$. Process B is translatable, as $Channels(B) \cap (\{\!|\ cs\ |\!\})$ is non-empty. On the other hand, process C is non-translatable, as $Channels(C) \cap (\{\!|\ cs\ |\!\})$ is empty.

5. External Choice Actions can have only either Prefixing Action branches (that can be guarded by a predicate or not) or other External Choice Actions. In the case of External Choice Processes, only other External Choice Processes can be branches or Basic Processes whose main action is a prefixing action (guarded by a predicate or not);

6. Only Free Types are used as the indexing set for iterated operators. This limitation is due to the fact that $\mathbb{A}$ types are infinite;

7. Only Basic Channel Set Expressions are accepted by ***JCircus*** for defining Channel Sets. This requirement is a result from a limitation of the typechecker of *Circus*: it does not allow unification ($\cup$), intersection ($\cap$) or subtraction (with the set-minus operator $\setminus$) of channel sets. More information about the typechecker of *Circus* can be found at (Xavier et al. 2006) and (Xavier 2006);

8. Only Propositional Predicates (those with operators $\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, $\Rightarrow$ and predicate expressions) are accepted by ***JCircus***. Quantifiers ($\forall$ and $\exists$) are forbidden. This limitation would be resolved integrating ***JCircus*** with a theorem prover;

9. Multi-synchronised channels that communicate at least one value of type $\mathbb{A}$ cannot exceed the maximum chosen integer value by the user when executing ***JCircus***. This is a requirement that the user of ***JCircus*** must assure when choosing the maximum integer value (the other requirements are assured by the tool);

10. The only translatable Commands are If-Guarded-Commands, Variable Blocks and Assignments. Specification Statements and its syntactic sugarings do not have a

translation rule, as well as Substitution commands (**val**, **res** and **vres**). Indexed Processes are also not translatable, but can be refined to parameterised processes using their translation rule on (Freitas 2005)

11. Processes and channels with generic types (since Freitas' ***JCircus*** version (Freitas 2005)) do not have a translation of graphical user interface, so they are also not translatable. This is a limitation that ***JCircus*** always had, and is due to the fact that channels with generic types can communicate values of different types during the specification, and the type should be known by the generated GUI (what does not happen);

The translatable grammar of *Circus* on ***JCircus*** can be seen at appendix A.

4.4.3 The structure of the generated code

When the requirements for using ***JCircus*** are assured, the tool generates a code that is organized in classes and packages. The structure of the packages is shown as follows (figure 4.8). The code is divided in the following 6 packages:

1. axiomaticDefinitions: this package has a class that implements methods for each axiomatic definition;

2. ccmaps: this package has classes that implement part of the code for complex communications;

3. channels: this package has classes with absolute values for each channel;

4. processes: this package has the classes that implement the code for each process of the specification;

5. typing: this package has classes for all free types created on the specification;

4.4.4 Executing the generated code

For each process on the specification given as input, ***JCircus*** generates, among other classes, a Java class that implements its behaviour and a Graphical User Interface (GUI) that allows a direct interaction between the user and the process. The GUI of each process is composed by buttons, each of which corresponds to a channel of the process' interface (visible channels). These buttons may be followed by combo boxes, each of

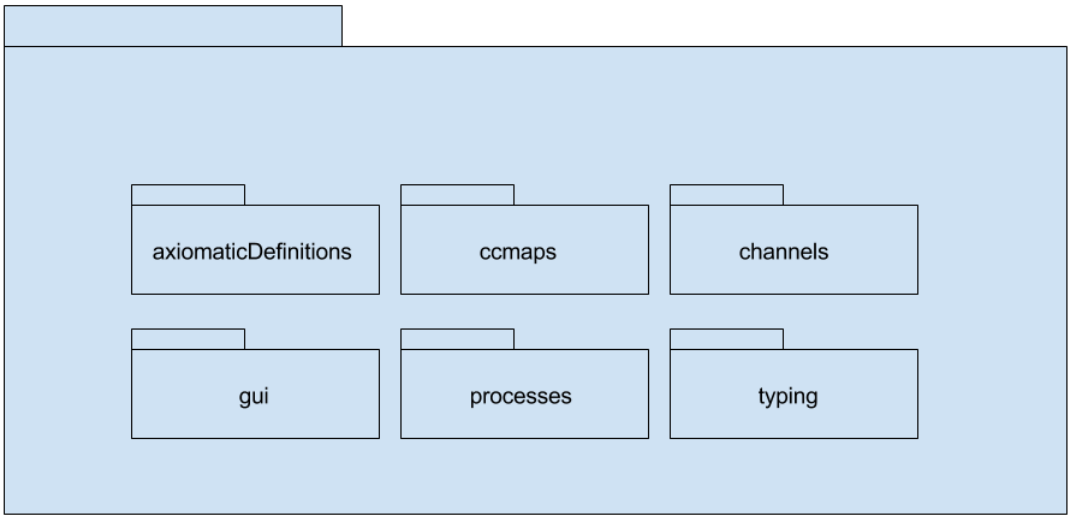

Figure 4.8: The structure of packages of the code that ***JCircus*** generates

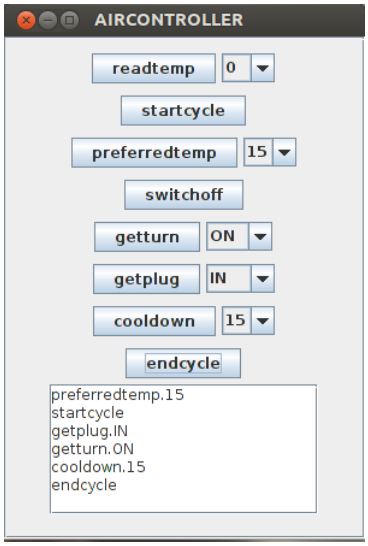

Figure 4.9: Graphical User Interface for the *AIRCONTROLLER* process

which corresponds to a communication field of the channel. Using this GUI, the user may interact with the generated code. For example, Figure 4.9 illustrates the GUI of the *AIRCONTROLLER*. Each button corresponds to a visible channel of the process: *readtemp*, *startcycle*, *preferredtemp*, *switchoff* , *getturn*, *getplug*, *cooldown* and *endcycle*. The channel *gettemp*, however, is hidden from the external environment and is not part of its GUI. Each button may be associated with combo boxes that correspond to communication fields. When a button is clicked, a successful synchronisation on the corresponding event is logged in the text area. If the event is not being offered, nothing happens.

In our example, illustrated in Figure 4.9, an acceptable sequence of events is: *preferredtemp*.15; *startcycle*; *getplug.IN*; *getturn.ON*; *cooldown*.15; *endcycle*.

Chapter 5

Verifying *JCircus*

We created and implemented, for this thesis, an approach to verify ***JCircus*** by checking if the generated code by ***JCircus*** really implements the behaviour of its input specification. The approach includes the implementation of a toolchain that model checks the generated code by ***JCircus*** in order to verify if it refines the semantics of the input specification. The toolchain we implemented for this thesis includes a Labelled-Predicate Transition System Generator (LPTSGen) for *Circus* (based on the Operational Semantics of *Circus*, whose soundness with respect to the UTP is under proof by this thesis) and a JPF Model Generator.

5.1 Basic Concepts

Before explaining the toolchain, we will explain some basic concepts: Java Pathfinder (JPF) and EclEmma.

5.1.1 Java Pathfinder (JPF)

Java Pathfinder (JPF) (NASA 2005) is a code model checker for Java bytecode. Its first version was developed by NASA Ames Research Center and since NASA open-sourced it was extended and used for other purposes than model checking. With JPF it is possible to execute a Java program not only once, but in all possible ways, searching for properties and checking deadlocks.

The major features of Java Pathfinder are:

- Choice Generators: this is a set of methods (from class Verify) that, when being run on the Java program under JPF, bifurcates the program in different choices according to a given interval on a given type. Examples of Choice Generators

are: `Verify.getInt(int, int)`, `Verify.getBoolean(boolean, boolean)`, and etc;

- Search and VM Listeners: these features are Java listeners that allow the interception of the model checking under JPF. On the case of VMListener, the instructions of the Java bytecode can be intercepted;
- Model Java Interface (MJI): in some cases, it can be desirable that JPF is run only for some parts of the program, but not for all parts. MJI has a set of constructs that allows the execution of parts of the program natively, that is, outside the reach of JPF;
- Bytecode Factories: this feature allows changing the mode of execution of each bytecode instruction;

Figure 5.1 was extracted from (NASA 2005) shows the functionalities of Java Pathfinder. We give, as follows, a simple example of a Java class whose method returns the sum of

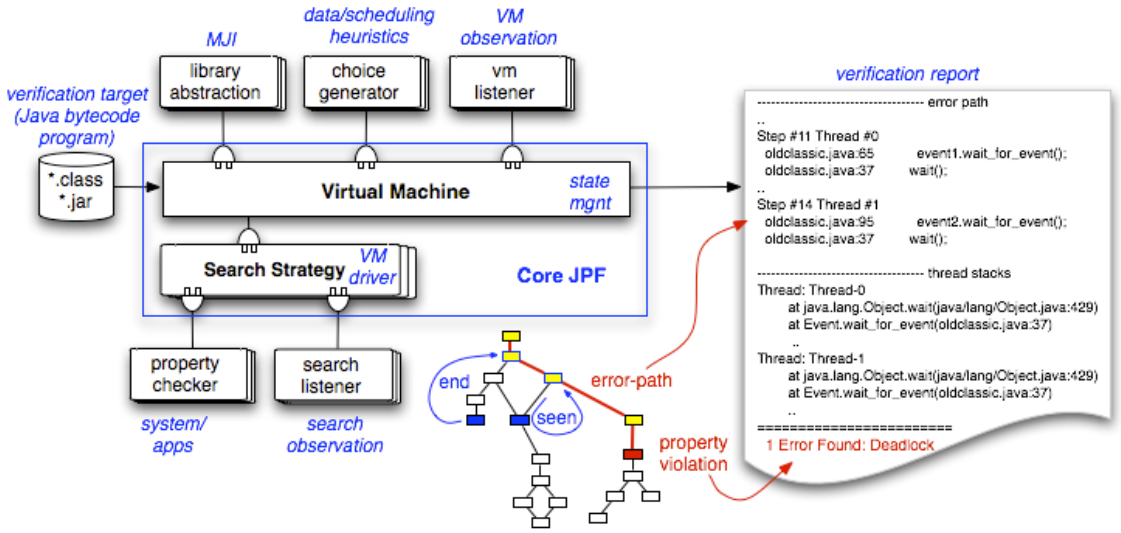

Figure 5.1: Java Pathfinder functionalities with its extensions and features

two integer numbers:

```
public class SumIntegers {
        int x, y;
        public SumIntegers (int x, int y) {
                this.x = x;
                this.y = y;
        }
        public int sum (int x, int y) {
```

```
                return x + y;
        }
        public int sum () {
                return sum (this.x, this.y);
        }
}
```

If we want to model-check the program `SumIntegers` in order to execute it for given ranges of `x` and `y`, we must create a JPF model (implemented as a Java program) that instantiates `SumIntegers` and generates choices for `x` and `y` using JPF methods. The following Java program (`SumIntegersModel`) shows the JPF model.

```
import gov.nasa.jpf.vm.Verify;
public class SumIntegersModel {
        public static void main (String [] args) {
                int d = Verify.getInt(1, 3); /*Instruction (1)*/
                int e = Verify.getInt(1, 3); /*Instruction (2)*/
                SumIntegers si = new SumIntegers (d, e);
                System.out.println (d + " + " + e + " == " + si.sum());
        }
}
```

Instructions (1) and (2) from `SumIntegersModel` show the use of choice generator `Verify.getInt(int, int)` for variables `d` and `e` in order to generate possibilities of execution for range [1, 3] twice. Figure 5.2 shows the bifurcation of the execution that JPF generates when it finds the choice generators. From figure 5.2 we can see that the use

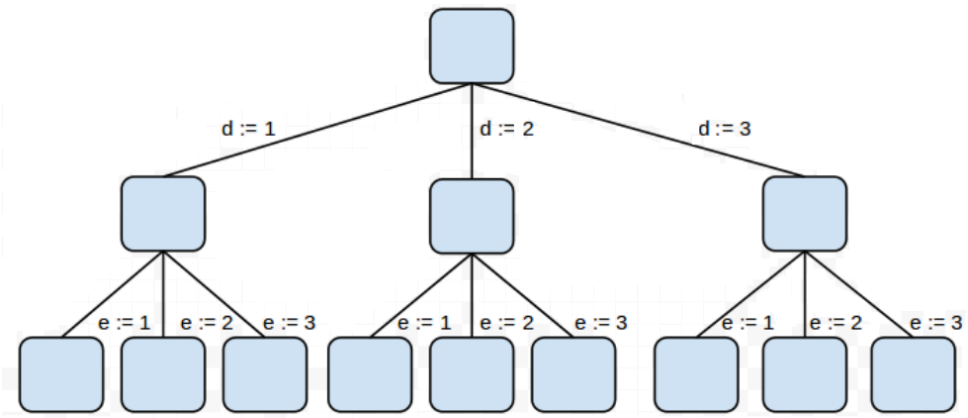

Figure 5.2: Figure showing the bifurcations that the Verify.getInt generator makes

of the choice generators on the model, when run through JPF, generates 9 possibilities of execution. When JPF reaches an instruction formed by a choice generator, it bifurcates and then continues the execution for the leftmost non-executed path. When it reaches the bottom of the program, it backtracks until it finds a bifurcation of execution, and then continues the execution for the leftmost non-executed path again. The process continues until it reaches the bottom of all possibilities of execution.

There are cases where it is desirable to use JPF only in some parts of the program, but not on the whole program being model-checked. When JPF is used for classes that uses threads, for example, JPF enters the code with threads and that leads the number of cases being model-checked to an undesired explosion. As the JCSP API was implemented on top of threads, the use of JPF with JCSP may not be desirable. Now, consider the following *Circus* specification:

$$\textbf{process}\, JCSPExample \mathrel{\widehat=} commvalue : \mathbb{N} \bullet \textbf{begin} \bullet o2o!commvalue \rightarrow Skip \,[\![\, \{\!|\, o2o \,|\!\} \,]\!]\, o2o?x \rightarrow Skip\, \textbf{end}$$

it can be encoded, using JCSP, as:

```
public class JCSPExample {
        private static AltingChannelInput aci;
        private static AltingChannelOutput aco;
        private static int commvalue;
        public JCSPExample (
                AltingChannelInput aci
                , AltingChannelOutput aco
                , int commvalue
        ) {
                this.aci = aci;
                this.aco = aco;
                this.commvalue = commvalue;
        }
        public static void execpar () {
                CSProcess csp1 = new CSProcess () {
                        public void run () {
                                aci.read();
                        }
                };
```

```
                CSProcess csp2 = new CSProcess () {
                        public void run () {
                                aco.write (commvalue);
                        }
                 };
                (new Parallel (new CSProcess [] {csp1, csp2})).run();
        }
}
```

On program `JCSPExample`, method `execpar()` encodes the behaviour of the main action of process `JCSPExample`. Now let's encode, using Java Pathfinder, a model that verifies the execution of *JCSPExample* for *commvalue* $\in$ [0, 1000], called `JCSPExampleModel`. As we don not want JPF to enter the JCSP code, it is necessary to execute the JCSP code under a native layer. That is where Java Model Interface (one of the features of JPF) takes place. If we want to create a native layer for class `JCSPExampleModel`, we must name it `JPF_JCSPExampleModel`. The `JPF_` prefix is necessary to indicate JPF that the class is a native layer. The following source code shows the implementation of the native layer.

```
import org.jcsp.lang.Channel;
import org.jcsp.lang.One2OneChannelSymmetric;
import gov.nasa.jpf.annotation.MJI;
import gov.nasa.jpf.vm.MJIEnv;
import gov.nasa.jpf.vm.NativePeer;
public class JPF_JCSPExampleModel extends NativePeer {
        @MJI
        public void executeNatively (MJIEnv env, int objref, int commval) {
                One2OneChannelSymmetric o2o
                        = Channel.one2oneSymmetric();
                JCSPExample je =
                        new JCSPExample (o2o.in(), o2o.out(), commval);
                je.execpar();
        }
}
```

The prefix `JPF_` on the name of the class, the annotation `MJI` and the extension to `NativePeer` indicate that JPF is going to run this part of the program natively, that is, using the native Java Virtual Machine, instead of JPF. Methods that will be invoked natively (like `executeNatively`) have two mandatory parameters: an `MJIEnv` parameter

(that is executed by the Virtual Machine of Java and guards the environment of the native method that has it as parameter) and an integer that guards the reference of the method. Parameters that come after these two are method parameters, and have to be redeclared on the definition of the method on the non-native class. The non-native class `JCSPExampleModel` is defined as follows:

```
import org.jcsp.lang.Channel;
import org.jcsp.lang.One2OneChannelSymmetric;
import gov.nasa.jpf.vm.Verify;
public class JCSPExampleModel {
        public static native void executeNatively (int commval);
        public static void main (String [] args) {
                int commval = Verify.getInt (0, 1000);
                executeNatively (commval);
        }
}
```

From class `JCSPExampleModel` we can observe that there is a declaration of a method called `executeNatively` with parameter `int commval`. The parameter is the same as the one on `JPF_JCSPExampleModel`. What the model above does is to generate choices on range [0, 1000] and put the generated value as parameter of `executeNatively`. This method is declared here as `native` and its implementation is on `JPF_JCSPExampleModel`.

For executing the model, the user can either define a file with extension ".jpf" and use a mandatory file called "jpf.properties" (with the default properties chosen for model-checking through JPF). The ".jpf" file contains the classpath for all libraries whose constructs were used throughout the JPF model and the name of the target model. An example of definition of ".jpf" file for class `SumIntegersModel` is:

```
+classpath=${config_path}/bin;/home/samuel/Softwares/Juno/eclipse/plugins/
                org.aspectj.runtime.source_1.7.3.20130613144500-a.jar
target = SumIntegersModel
```

The JPF user can also run JPF embedded in regular Java. The following Java code shows how that can be done.

```
public class EmbeddedSumIntegers {
        public static void main(String[] args) {
                String[] args2 =
```

```
                        new String[] {
                                "+classpath=${config_path}/bin;"
                };
                Config config = JPF.createConfig(args2);
                config.setProperty("vm.por.sync_detection", "false");
                config.setProperty("cg.enumerate_random", "true");
                config.setProperty("cg.threads.break_sleep", "false");
                config.setProperty("vm.por", "false");
                config.setProperty("search.multiple_errors", "false");
                config.setProperty("vm.por.fieldlockinfo.class", "false");
                config.setProperty("native_classpath", "bin");
                config.setTarget("SumIntegersModel");
                JPF jpf = new JPF(config);
                jpf.run();
        }
}
```

When running JPF embedded, properties of JPF are defined via method `setProperty`, from `Config`, and the classpath is defined via an array of strings of paths. The model to be model-checked is chosen by invoking `setTarget`, and JPF is set to run instantiating an object of type `JPF` and invoking `run`.

Java Pathfinder has some limitations that are listed as follows:

- Limited support for model-checking Java Graphical User Interface constructs;
- Limited support for storing information about the model-checking;
- Limited possibility of interaction between native and non-native layers on model checking (only variables of primitive types can be passed from the native to the non-native layer);

5.1.2 EclEmma

EclEmma (Hoffmann 2006) is an Eclipse plug-in that measures code coverage for a Java program that is executed. The code coverage on a program indicates what is the percentage of that program that was executed. The classical coverage measures are:

- Statement coverage: has each statement (instruction) been executed?
- Decision coverage (DC): has each statement (instruction) been executed and have all possible values (true and false) of all if-then-elses been tested?

- Modified Condition/Decision coverage (MC/DC): has it Decision coverage and all the boolean variables of each if-then-else statement influence the output?
- Multiple Condition Coverage: has it Decision coverage and were all combinations of values for the boolean variables of each if-then-else statement tested?

What EclEmma does is to give the percentage of DC (Decision Coverage) that is achieved for its code when the program is run. All the fully visited statements are painted as green, all the unvisited statements are painted as red, and the statements that are partially visited are painted as yellow. The statements are considered as fully visited when MC/DC is reached. Consider the following code.

```
public class CovTest {
        public static void covcov (int x, int y, int z) {
                if (x == 1 || y == 1 || z == 1)
                        System.out.println ("Came inside the if statement");
                System.out.println ("Now outside the if statement");
        }
        public static void main (String [] args) {
                new CovTest (); //Line 8
                int x = 1, y = 0, z = 0; //Line 8
                covcov (x, y, z); //Line 9
                x = 0; //Line 10
                covcov (x, y, z); //Line 11
                /*y = 1; //Line 12
                covcov (x, y, z); //Line 13
                y = 0; //Line 14
                z = 1; //Line 15
                covcov (x, y, z);*/ //Line 16
        }
}
```

When the above program is run on Eclipse using EclEmma, 100% of code coverage (Decision Coverage) is achieved, because the expression on the `if` statement is leaded to both `true` and `false` by calling `covcov` twice and varying the value of `x` between these calls. As the other branches of the `if` statement were left unchanged, MC/DC coverage did not reach 100%. Figure 5.3 shows the percentage of DC achieved on the program above and the painted code (according to the coverage achieved for each statement).

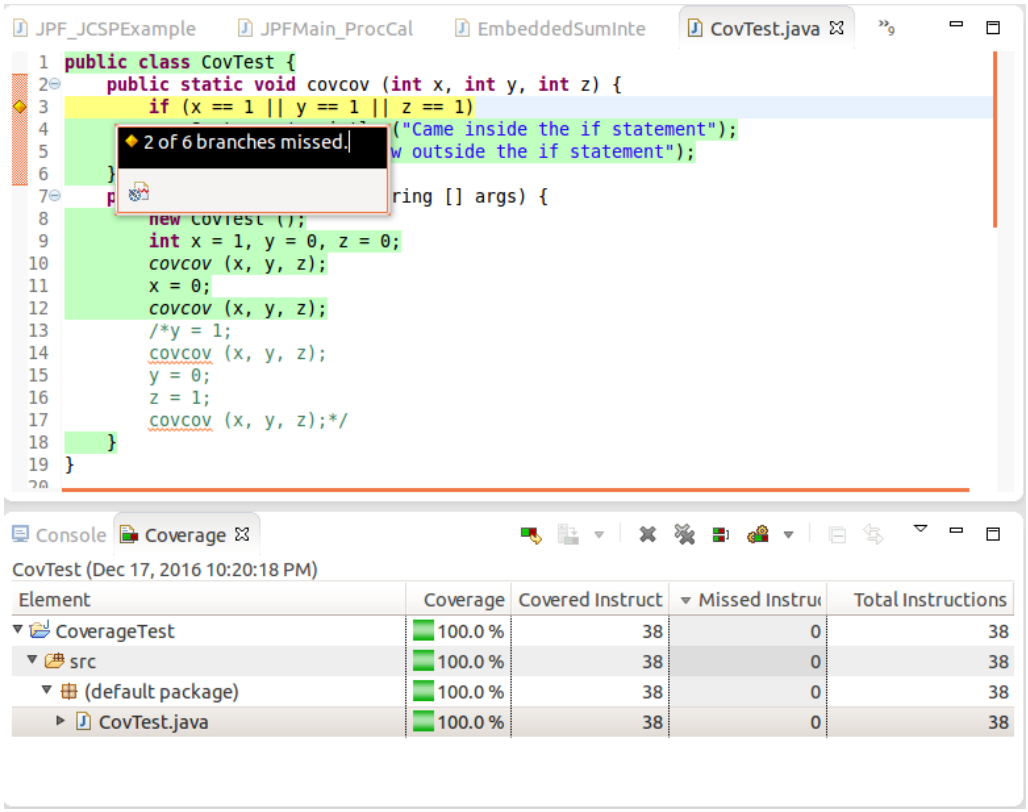

Figure 5.3: Screen showing branches that were missed during coverage measurement

If we removed the $/*$ and the $*\backslash$ on lines 12 and 16, DC is achieved and all the three branches `x == 1`, `y == 1`, `z == 1` are proved to be capable of individually affect coverage, what also makes 100% MC/DC.

5.2 The strategy of verification

On this section we will explain the strategy we created and used for verifying the generated code from *Circus* to Java using ***JCircus***. It is divided in two major parts:

- Implementation of a Toolchain for model-checking the generated code by ***JCircus***, in order to check if the generated code by ***JCircus*** refines the input specification. The toolchain encompasses a Labelled-Predicate Transition System Compiler for *Circus* (LPTSGen) and a JPF Model Generator;
- Execution of ***JCircus*** together with the Toolchain for an input set of *Circus* specifications (written in LaTeX) with high code coverage for ***JCircus***;

Figure 5.4 illustrates the strategy:

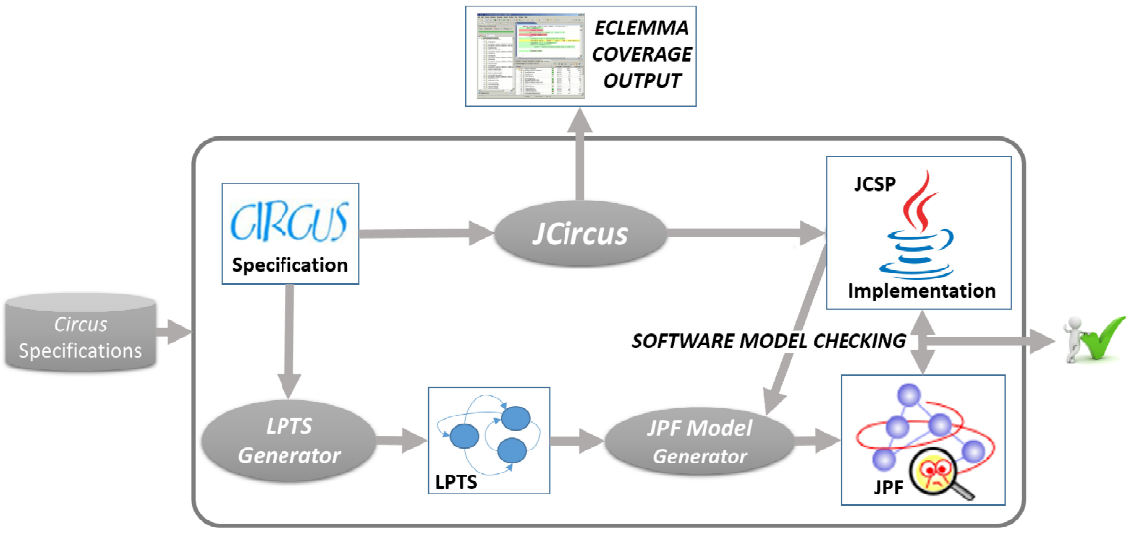

Figure 5.4: Strategy for verifying ***JCircus***

We saw, on the Introduction 1, that the Strategy for verifying ***JCircus*** (shown in figure 5.4) encompasses an input set of *Circus* **specifications** submitted one by one (*Circus* **specification**) as input both for ***JCircus*** and for the **LPTS Generator**. Each *Circus* **specification** submitted as input for ***JCircus*** increases the code coverage from ***JCircus*** measured by EclEmma, and EclEmma produces the **ECLEMMA COVERAGE OUTPUT**, that shows the percentages of Decision-coverage for each package of the **source** code of ***JCircus***. ***JCircus*** also generates Java code (**JCSP Implementation**) that implements the specification. When the *Circus* **specification** is submitted as input for the **LPTS Generator**, the LPTS Generator generates the **LPTS**. Both the **LPTS** and the **JCSP Implementation** are submitted as inputs for the **JPF Model Generator**. The **JPF Model Generator**, then, generates the **JPF** Model: this model performs the **SOFTWARE MODEL CHECKING** that calculates the semantic equivalence between the *Circus* **specification** and the **JCSP Implementation**.

From the modules shown in figure 5.4, only ***JCircus*** and EclEmma were previously implemented. The other modules (the **LPTS Generator** and the **JPF Model Generator**) were implemented for this PhD thesis. The *Circus* **specifications** input set was manually constructed.

We will begin explaining the Labelled-Predicate Transition System Generation for *Circus* specifications on sub-section 5.2.1.

Each of these steps will be detailed on subsections 5.2.1 and 5.2.2.

5.2.1 The Labelled-Predicate Transition System Generation for *Circus* specifications

We implemented, for this thesis, a Labelled-Predicate Transition System Generator (LPTSGen) for *Circus*, based on the Operational Semantics described in 3.1.2. The LPTSGen has two similar modules of ***JCircus***: the parser and the typechecker, that were implemented by CZT. After parsing and typechecking the specification, the AST (Abstract Syntax Tree) generated by the parser will then be compiled (by the third module, the LPTS Compiler) into a Transition System that describes a graph-like structure. The implementation of the LPTS Compiler was guided by the structure described in (Freitas 2006) and by the Operational Semantics rules described in 3.1.2. Figure 5.5 shows the process of generation of the LPTS. The implementation of the LPTS Compiler was made

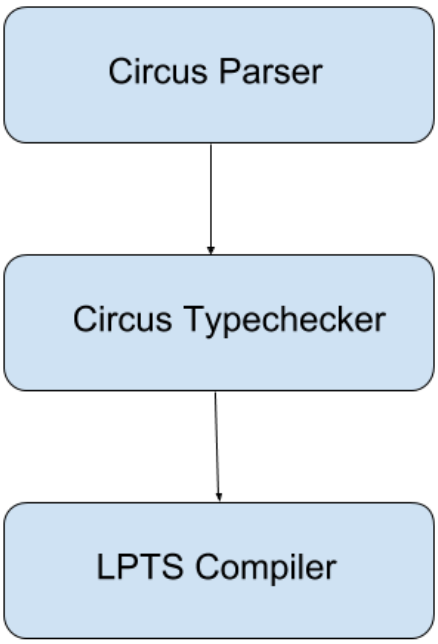

Figure 5.5: Figure showing the modules of the LPTSGen

by using the Visitor Design Pattern (Vlissides et al. 1995) of CZT, in which the logic for each syntactic category was implemented using its corresponding visitor. For example: for Variable Block, `VarDeclCommandVisitor` was used to allow the implementation of the visitor `visitVarDeclCommand`, that has the logic of the construction of the Transition System for Variable Blocks. Figure 5.6 shows the structure of packages for LPTSGen. Each of the packages are:

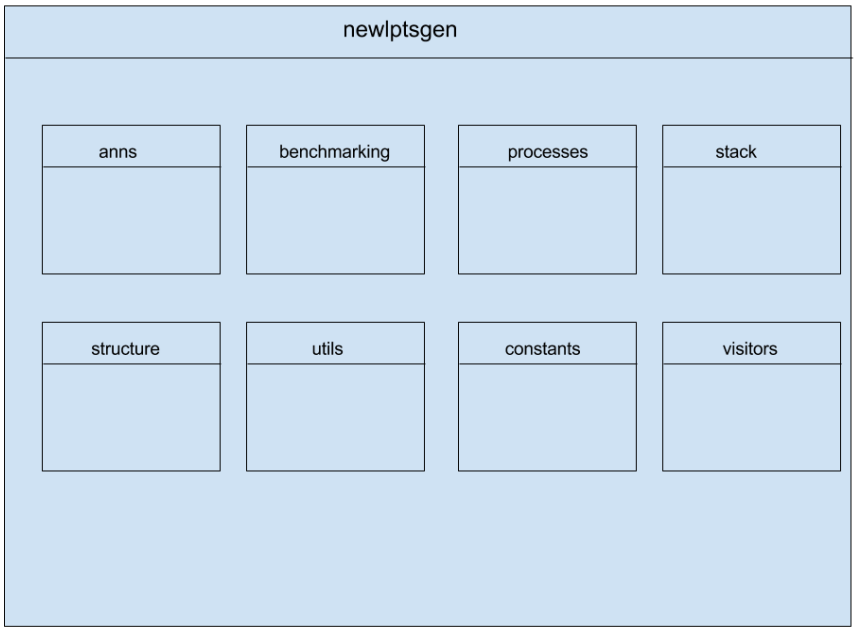

Figure 5.6: Figure showing the packages of the LPTSGen

- newlptsgen.anns: has classes used for annotating the AST with information about the LPTS. For example: the variables on the partition of the parallelism;
- newlptsgen.benchmarking: this package has classes that allow the execution of LPTSGen for a given number of specifications;
- newlptsgen.processes: this package has classes with methods that are used for the application of the Operational Semantics rules for *Circus* processes;
- newlptsgen.stack: this package has a class that implements a scope stack for the LPTS. This scope stack stores information about the current constraint, the current state, the current program text, and etc;
- newlptsgen.structure: this package has classes that implement the data structures for the LPTS: `Node`, `Constraint`, `LPTSState`, `Transition`, `TransitionSystem` and etc;
- newlptsgen.utils: this package has classes with utilitary methods for the LPTS construction. It encompasses methods for merge of external choice and parallel branches, for printing the LPTS, and etc;
- newlptsgen.constants: define constants that will be used for printing the LPTS;
- newlptsgen.visitors: implements the construction of the LPTS for *Circus* actions using the Visitor Design Pattern of CZT;

As it was seen on 3.1.2, a rule is formed by a condition and a transition relation:

$$\frac{P}{Trans}$$

What the relation above means is that the transition rule (denoted by *Trans*) is valid only if predicate *P* is true. There are three kinds of factors (sub-expressions) that predicate *P* can contain:

- Runtime factors: these factors can either involve a constraint *c* that can appear on the nodes of the transition and state and boolean factors involving loose constant values. For example: the rule of Assignment 1, that has *c* and $(s;\ (w0 = e))$ as factors;
- Type factors: these factors involve conditions related to type trustworthiness: In the case of Prefixing Input 2, there is a Runtime factor *c* and other two type factors: T $\neq \emptyset$ and x $\notin \alpha$(s);
- Branch factors: these factors involve transitions for compound operators whose predicate P depends on conditions of the LPTS of the branches of the action. For example: External Choice Silent (left 16 and right 17), External Choice End 15 and Parallel Independent (left 20 and right 21);

Type factors are handled by the typechecker of *Circus*, so if a type factor on predicate *P* is not satisfied, the typechecker prohibits the construction of the LPTS. Branch factors are handled during the construction of the LPTS: in the case of branch factors, the constructed LPTS inherits the LPTS of its branches, so the LPTS for the compound operator is constructed by re-using the LPTS of its branches (both External Choice and Parallel operators have *merge* methods). Runtime factors are handled after the construction of the LPTS, by the JPF Model Generator (which will be explained afterwards).

As it was seen on 3.1.2, each transition is formed by a source node, an arc and a destination node. Each node has a Constraint, a sequence of Assignments and a Program Text. Figure 5.7 shows a class diagram for some classes of the LPTS Compiler. As we can see from figure 5.7, `Transition` is a class that has a `Node` object representing the source node, an `Arc` object representing the arc, and another `Node` object representing the destination node of the transition. `Node` is a class that has a name (`String`), a `Configuration` (that encapsulates the sequence of assignments, the Constraint and the Program Text) and an environment (`LPTSEnvironment`). The `Arc` is a class that contains the CZT term `Communication`, a channel name (of type `String`) and a vector of vector of strings with loose constant sets. Here, the loose constants are grouped as sets in order to help the generation of the JPF Model (which will be explained on the following subsection), because

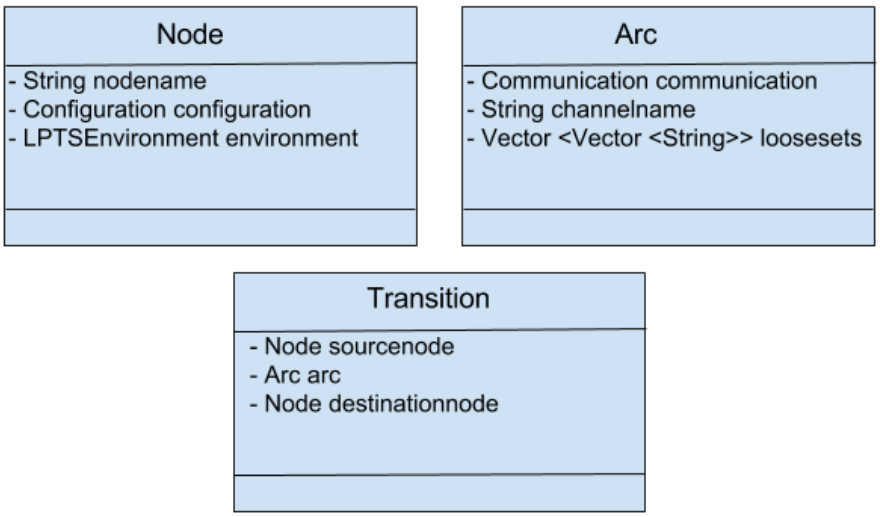

Figure 5.7: Figure showing some classes of the LPTS Compiler

some loose constants will have to be tested with similar values. Those loose constants with similar values will be grouped on the same set.

Other elements of the LPTS, like the local copies *loc* on Program Texts, are infered on the scope stack as the state on the top of the stack. The partitions on the parallelism (rule Parallel Begin 18) are stored on the scope stack as variables *barvars* and *plusvars*. The rules of Hiding are implemented by stacking the hidden channels on the scope stack and ”silencing” (converting to τ) the labelled transitions whose arcs have the same name of a channel on the channel set of the hiding. The *let* construct on Variable Block Begin 4 does not lie on the syntax of *Circus*, but has the role of creating an intermediate state where the declared variables are on the scope of the action. This effect is achieved by storing the variable names (and their types) in a hash map and stacking the scope. Construct *end* (from Variable Block End 6), is another one that does not lie on the syntax of *Circus*, and the effect of this construct is achieved by unstacking the scope when the action finishes.

In the following subsection, we will explain the JPF Model Generator.

5.2.2 The Java Pathfinder Model Generation

We saw, on section 5.2.1, that the LPTSGen receives a *Circus* specification as input (written in LaTeX *Circus*) and generates a Transition System with transitions between nodes and arcs. What the JPF Model Generator does is to get as input a Transition System and a JCSP code generated by ***JCircus***, and returns a Java Pathfinder code that model-checks (checking both acceptances and refusals) the JCSP code generated by ***JCircus***. Figure 5.8 shows both LPTSGen and JPF Model Generator.

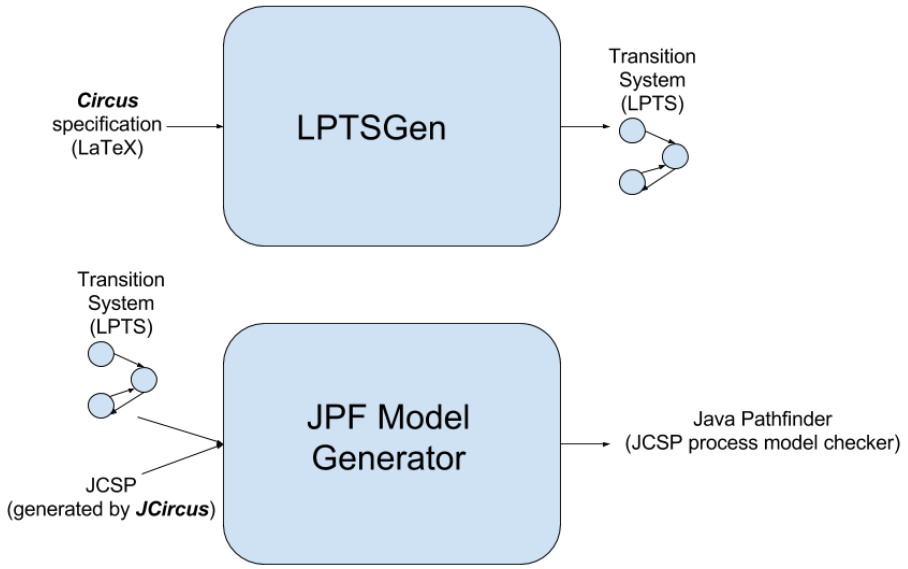

Figure 5.8: Processes of generation of the toolchain (LPTSGen + JPF Model Generator)

Conducting the generated code of *JCircus* for a given specification: how to manually prove that it refines the specification?

Consider the following *Circus* process:

process P $\widehat{=}$ **begin** • a.0 $\rightarrow$ b.1 $\rightarrow$ Skip **end**

Consider that the chosen range of integer numbers is [0, 1]. Thus, the set of events Σ equals $\{$a.0, a.1, b.0, b.1$\}$. Now, consider a given process Q. In order to check if Q $\sqsubseteq$ P in the failures-refinement model, what we must check is if Q:

- initially rejects a.1, b.0 and b.1 and accepts a.0;
- after accepting a.0, rejects a.0, a.1 and b.0, and accepts b.1;

The checking that the generated code refines the input specification can be done by the user of ***JCircus*** by running the GUI of the generated code of process P and testing the sequence above. If, at the end of the test, a.0 and b.1 appeared on the text area (and no other event appeared), then the user is able to know that the generated code by ***JCircus*** really

implements the input specification. Figure 5.9 shows this.

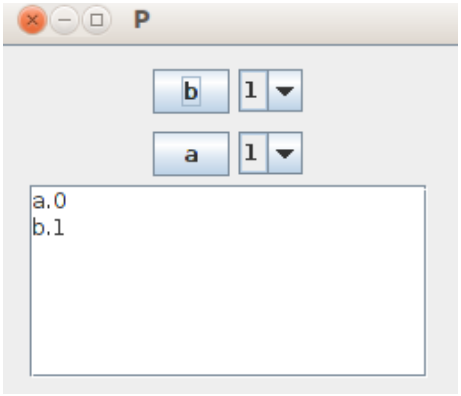

Figure 5.9: The GUI of ***JCircus*** for the test above

The decision of what event to accept and what event to refuse at any state can be guided by the LPTS (figure 5.10) of the specification: all labelled arcs whose destination node constraints evaluate to true must correspond to acceptable events. Otherwise, they must be rejectable events. Thus, in the case of our example, from n1 to n2, the only event that

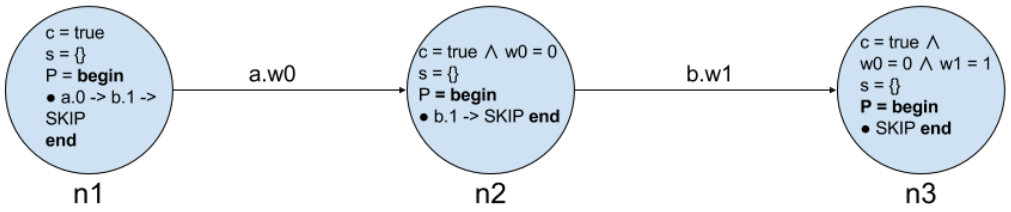

Figure 5.10: The LPTS for the specification that was used as example

is correspondent to an acceptable event (that is, makes the constraint of n2 be true and is equivalent to arc a.w0) is a.0. All the other events must be rejected at this point. Thus, if we click (on the GUI of the generated code by ***JCircus***) a.1, b.0 and b.1 and they are refused by the process and a.0 is accepted, then the behaviour of the process is authentic to the one on the LPTS at this point. After the acceptance of a.0, the only event that must be accepted, according to the LPTS, is b.1. If that happens on the generated code by ***JCircus***, then the semantics of the LPTS is refined by the generated code by ***JCircus*** for the exercised specification. On the following subsection we will formalise the strategy and focus on it.

The strategy from the LPTS to Java Pathfinder (JPF)

JPF Model Generator generates a JPF model that performs similarly to the process described previously, having the clicks on the buttons replaced by the direct invocation

of each click's behaviour (as JPF does not work well for AWT). The strategy, however, becomes more sofisticated for more complex processes, where nodes can have more than one arc coming out (external choice and parallel compositions make this) or where non-determinism occurs (through non-single silent transitions, caused by internal choices or hiding).

JPF Model Generator exercises all the transitions of all paths of execution. Basically, branches are originated either from choices (internal or external) or from the generation of values for communication, which are represented as loose constants in the LPTS. The treatment of non-τ transitions is different from the treatment of τ transitions. The non-τ transitions are used to build the set of accepted events and the set of refused events at each source node. The former is composed of events whose transitions lead to destination nodes with constraints evaluated to true, and the latter is composed of events that either have no transitions leaving the node or whose transitions lead to destination nodes whose constraints evaluate to false. If either a click on an event is accepted and this event is in the set of refused events or a click on an event is refused and this event is in the set of accepted events, the refinement fails. Otherwise, the exercise continues by examining the target nodes. The τ transitions do not cause any event click. In these cases, we just check the constraints of the destination nodes. If any of them fails, the refinement fails. Otherwise, the exercise continues by examining the target nodes. This examination continues until no further non-examined destination nodes are found. Finally, the refinement is successful if, and only if, it is successful for all paths. Figure 5.11 shows the general structure that can be found on a node on an LPTS.

Consider that $Ref(n_k)$ is the refinement for node n_k on figure 5.11. Also, consider that l_{n+1} to l_{n+t} are labels of arcs that do not belong to n_k. At last, let $acc(l)$ be the boolean function that tests if event l is accepted and $constr(n)$ be the constraint of node n. Also consider the following formulas (based on figure 5.11):

$$S(n_k) = \exists a : [k+1, k+i].constr(n_a) \wedge Ref(n_a) \tag{5.1}$$

The formula 5.1 calculates, for an a varying from $k+1$ to $k+i$, if any of the destination nodes n_a has successfull refinement. For that to happen, there might exist at least one destination node with constraint $constr(n_a)$ being true and refinement $Ref(n_a)$ successfull. The nodes whose indexes go from $k+1$ to $k+i$ are the destination nodes of figure 5.11 that are reachable throught silent (τ) transitions. Thus, $S(n_k)$ calculates if the refinement is successfull for at least one of the destination nodes reachable from n_k throught silent

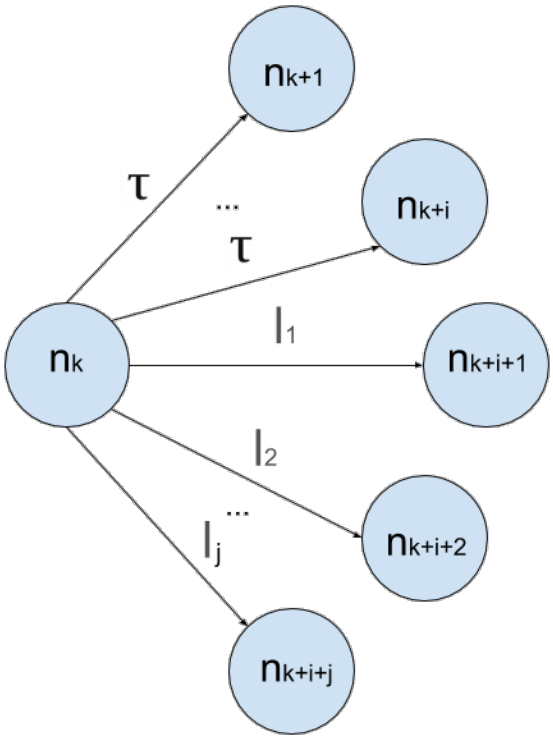

Figure 5.11: General representation of a node with its arcs and destination nodes

transitions: that might be the case when there is non-determinism. Now we define formula

$$L(n_k) = (\forall b : [1,j].acc(l_b) \wedge constr(n_{k+i+b}) \wedge Ref(n_{k+i+b})) \wedge \forall c : [n+1,n+t].\neg\, acc(l_c) \quad (5.2)$$

The formula 5.2 calculates if (1) all events l_b whose source nodes are n_k are accepted having their destination node constraints ($constr(n_{k+i+b})$) being **true** and ($\wedge$) destination node refinements $Ref(n_{k+i+b})$ successfull (**true**), and (2) all the other events whose arcs do not come from n_k are refused ($\forall c : [n+1,n+t].\neg\, acc(l_c)$)

$$Ref(n_k) = S(n_k) \vee L(n_k) \quad (5.3)$$

Function $Ref(n_k)$ on formula 5.3 indicates that a refinement is successfull (**true**) if either (1) at least one of its destination nodes have successfull refinement having its constraint being **true** ($S(n_k)$) or (2) all events of arcs going out from n_k are accepted having their destination constraints **true** and their destination node refinements also **true** and all the other events are refused ($L(n_k)$).

The presence of non-determinism on the generated code by ***JCircus*** imposes that the refinement must be checked for all non-deterministic branches of the generated code. Let

BRANCHES be the set of non-deterministic branches of the generated code by ***JCircus***. Also, let $n1$ be the initial node of the LPTS of the specification given as input. The specification is considered succesfully refined by the generated code by ***JCircus*** if the following formula evaluates to true:

$$Refinement(Spec) = \forall\, branch : BRANCHES \,.\, Ref\,(n1) \tag{5.4}$$

The following sub-section shows how the process described on this sub-section was implemented.

The architecture of the JPF Model

The architecture of the implementation of our model checking strategy is presented in Figure 5.12. The arrows represent a relation between the JPF generator and the classes automatically generated: an arrow from a ***JCircus*** class to the JPF generator indicates that the class is used as input by the generator. For example, classes $Main\_\langle process\,name\rangle$, $Gui\_\langle process\,name\rangle$ and $\langle process\,name\rangle$ are used by the JPF generator. The inputs of the JPF Generator are the LPTS of the specification and these classes. A dotted arrow indicates that there is an aspect intercepting a class (in this case, $Aspect\_\langle process\,name\rangle$ intercepts class $\langle process\,name\rangle$). In what follows we describe, in more details, the role of each of the components.

The class $JPFDriver\_\langle name\_of\_the\_process\rangle$ implements the driver (conductor) of the code generated by ***JCircus***. This class is generated based on information in the LPTS: each node has a corresponding method in the driver that implements its behaviour. This behaviour includes the clicks on the events and the verification of its constraint, whose successful verification leads to a call to the methods that implement the behaviour of the subsequent nodes.

The class $Iterator\_\langle process\,name\rangle$ implements the iterator of our model checking strategy. The generation of this class is based on the GUI class of the process generated by ***JCircus***: it is a modified version of the GUI that removes the GUI elements. This class is used by the *JPFDriver* to interact with the translated code via methods, because, to the best of our knowledge, JPF does not work well for programs with GUI elements.

Next, the JPF Generator generates a class that provides the communication between the non-native and the native portions of the model checker code. The class $ModelCheck\_\langle name\_of\_the\_process\rangle$ also implements the generation of values, using `Verify.getInt`, for program paths (the a, the b and the c of functions $S(n)$ and $L(n)$ of $Ref(n)$), event

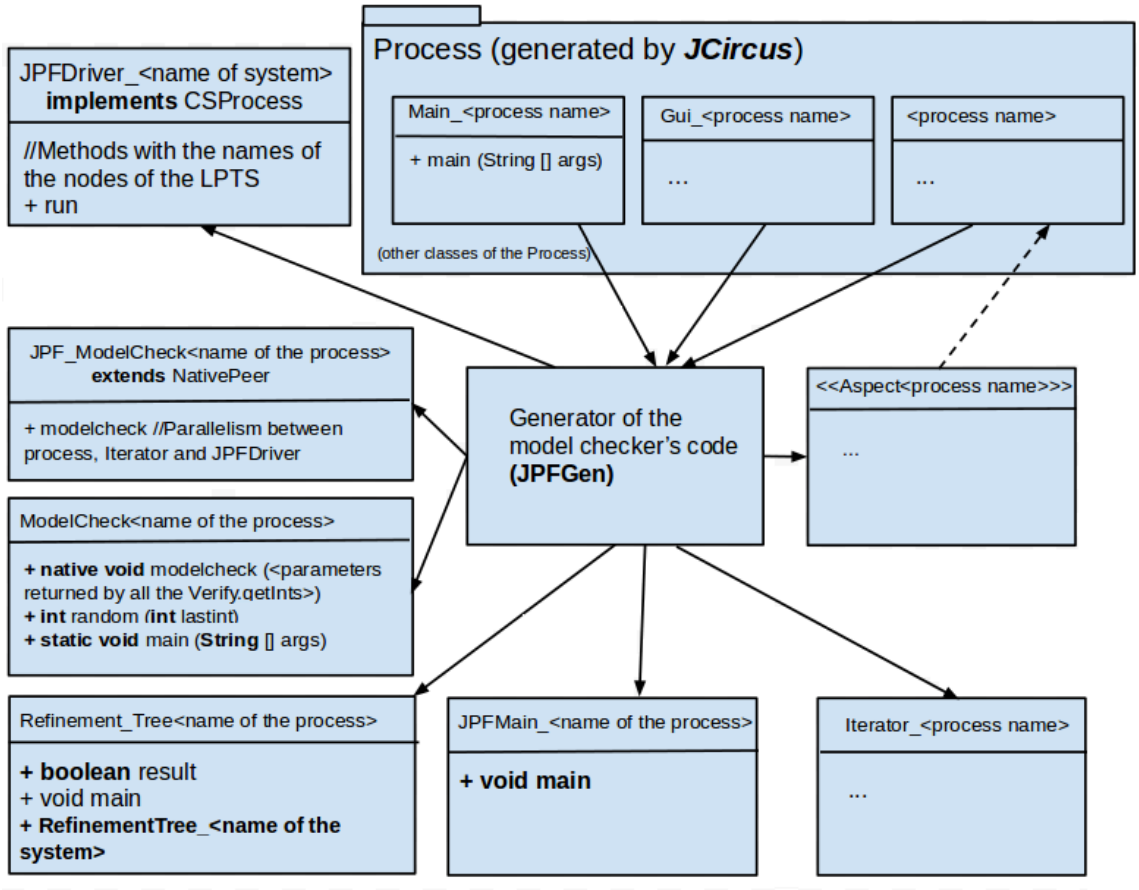

Figure 5.12: Architecture of the Refinement Strategy

clicks (from the alphabet Σ of events) and loose constants (when the arc is labelled and it communicates values). The generated values are given to the native portion of the model checker that we explain in the sequel.

The class $JPF\_ModelCheck\_\langle name\_of\_the\_process \rangle$ implements the native portion of the model checker. This class consists in a modified version of the main class, adapted to receive the generated values of event clicks and loose constants from the non-native part of the model checker. This class is executed natively by JPF (extends *NativePeer*). As a consequence, the internal behaviour of the JCSP classes are not model checked by JPF. The model checking of the JCSP classes would easily cause a state explosion. More importantly, though, it is not of our interest to verify the trustworthiness of the JCSP library. Our main interest is to model check the external behaviour of theses classes with respect to the *Circus* specification.

The process classes run natively in the model checker. For this reason, random instructions would also be executed natively, preventing us to check the process for all possible random values. This, of course, would violate the principles of our model

checking. In order to avoid this behaviour, we generate an aspect, $Aspect\langle process\,name\rangle$, that intercepts the process class generated by ***JCircus*** at the random instructions and diverts the control flow to the non-native part of the model checker.

Finally, the class $RefinementTree\_\langle name\_of\_the\_process\rangle$ implements a tree that maps the indexes of the execution paths of JPF to its leaf nodes, stores the result of the refinement verification of each path and checks if the set of results of each path corresponds to a successful refinement. This class overcomes the lack, in JFP, of functionalities to store data results.

5.2.3 The verification of *JCircus*

The criteria for the input-set

The strategy described so far is able to verify the generated code by ***JCircus*** for a given specification *Spec* and verify if the generated code refines the semantics of the input specification (that is, that ***JCircus*** produces reliable code for *Spec*). But how does that fit with the verification of ***JCircus***? If we apply this strategy to a satisfatory input set of specifications, we have a sofisticated testing strategy that is able to identify faults on ***JCircus***. Moreover, having the rules of the Operational Semantics proved with respect to the Unifying Theories of Programming (on the appendices of this document), we also link the generated code by ***JCircus*** with the UTP.

How do we choose a good criteria for the satisfiability of the input set? We considered, at first, using grammar-based coverage, like several works did (some examples are (Hentz et al. 2015, Lämmel & Schulte 2006)), but we realised that only grammar-based coverage would not be enought for the input set, because two specifications using the same syntactic categories can possibly exercise different parts of the source code of ***JCircus***. For example:

- Processes Par1 and Par2:

$$\textbf{process}\ Par1 \mathrel{\widehat=} \textbf{begin} \bullet a \rightarrow Skip \,[\![\, \{\!|\ a\ |\!\} \,]\!]\, a \rightarrow Skip\ \textbf{end}$$
$$\textbf{process}\ Par2 \mathrel{\widehat=} \textbf{begin} \bullet a \rightarrow Skip \,[\![\, \{\!|\ b\ |\!\} \,]\!]\, a \rightarrow Skip\ \textbf{end}$$

  Both processes, Par1 and Par2, use the same syntactic categories: both are process paragraphs (process N = ProcDef) whose process is a stateless basic process (**begin** • Action **end**) having a parallel action (CSPAction |[CSExp]| CSPAction) between two prefixings (Comm → Action) as its main action;

- Processes Hid1 and Hid2:

  $$\textbf{process}\, Hid1 \mathrel{\widehat=} Par1 \setminus \{|\ a\ |\}$$
  $$\textbf{process}\, Hid2 \mathrel{\widehat=} Par1 \setminus \{|\ b\ |\}$$

  Processes Hid1 and Hid2 also have similar syntactic categories: Call processes with a Hide Process.

Although both pair of processes have similar syntactic categories, their strategies exercise different parts of the source code of ***JCircus***: Par1 has a regular synchronisation between branches on a, whereas Par2 implements the forced interleaving strategy described on (Barrocas 2011). Hid1 has a hiding of an event that lies on process Par1, whereas Hid2 does not do that, and this makes each process exercise a different path on ***JCircus***' source code.

We considered as a satisfatory input set that one that made ***JCircus*** reach the higher possible Code Coverage. In order to achieve that, we determined that the input set (1) must firstly cover all the translatable grammar of *Circus*, and (2) cover all major conditions from the tool (some of which were quoted previously), which are: Single or multi-way synchronisation; Forced Interleaving or Parallel Synchronisation; Complex or simple communications; and Hiding with inner synchronisation or Hiding of event that do not occur on the hidden process.

The input-set and its verification

We constructed an input set that follows the coverage criteria explained on 5.2.3. This input-set encompassed 7 sub-sets of specifications for which the verification was made. For a given input sub-set, we considered that it verified ***JCircus*** when all the processes on its specifications had successfull refinements between the input of ***JCircus*** and the JCSP code on the output (see figure 5.4). On total, the time of execution of all 7 sub-sets was 14 minutes and 50 seconds. During the execution of the input-sets, two faults were detected, and will be explained at 5.2.3.

Sub-set index	Num. of Spec.	Num. of Proc.	Time of gen. + exec.
1	7	7	1:21
2	9	58	5:20
3	8	12	1:23
4	11	11	1:20
5	7	13	3:10
6	2	6	1:31
7	2	2	0:47
Total	45	109	14:50

The input sub-sets of the verification of ***JCircus***

Specifications were created and modified having (1) and (2) as a basis and the Decision Coverage (DC) was checked. Moreover, unused methods were removed (after the modifications of the extension to ***JCircus***, some methods became unused), and this process was repeated until the input set covered all reachable statements and all if-then-elses that depended on grammar conditions. We show, on figure 5.13, the percentage of DC achieved.

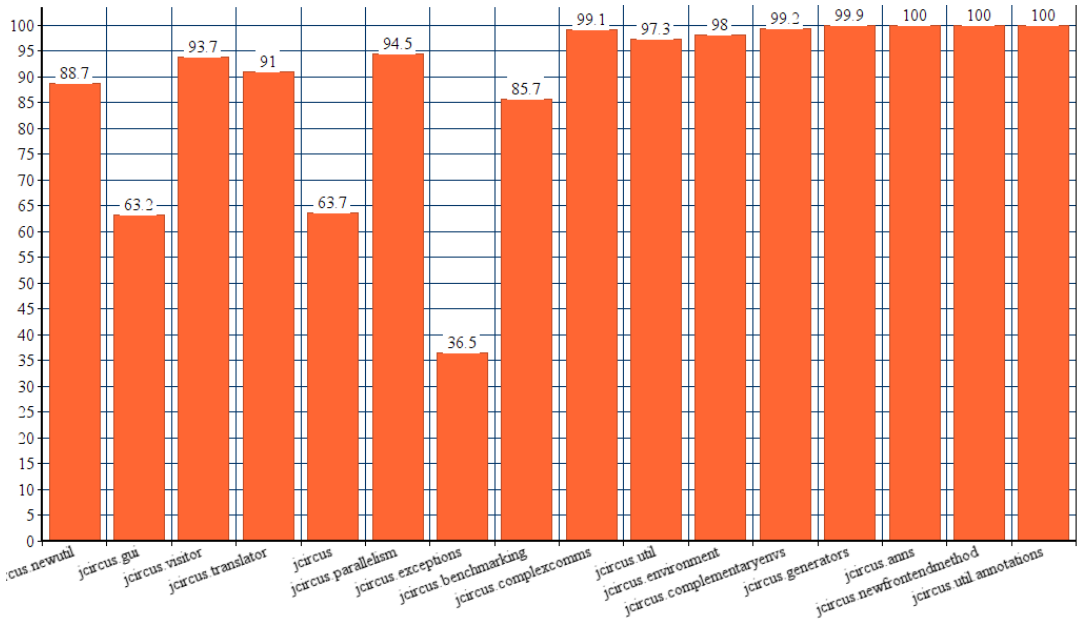

Figure 5.13: Coverage percentages of ***JCircus***, package by package

The non-covered percentage of the source code of ***JCircus*** refers to:

- Code parts that are executed only by LPTS/JPFGen, in the case of jcircus.newutil and jcircus.parallelism. As we reuse some libraries of ***JCircus*** that manipulate the

AST, we have syntactic categories that are only used by LPTSGen (like hide actions and alphabetised parallel actions). The syntactic categories not used by ***JCircus*** refer to non-translatable constructs;

- Code parts that are implementation exceptions (possible missing syntactic categories). Nevertheless, these exceptions only appear on parts of the code where there are non-expected syntactic categories (we left the exceptions for safety);
- Velocity exceptions on the creation of translation templates: Some parts of ***JCircus*** use VelocityContext for generating code (jcircus.visitor and jcircus.translator. These exceptions are not expected to be exercised because if they happened, it would indicate that there are missing templates for generating some parts of the code, which is not admissible;
- Code parts related to runtime exceptions (ChannelNotDefinedExceptionForProcess, for example, that checks if a given channel is not defined for a process. These are not expected by ***JCircus*** because if they happened, it would indicate malfunction of ***JCircus*** code;
- Unmarshal and IO exceptions; a throwed Unmarshal exception would indicate malfunction on the parser of *Circus* and an I/O exception would indicate that some input path was not found. So they are not expected;
- Branches from if constructs that test if a given object is null;
- jcircus.exceptions has very low coverage, but the package is very small. The non-covered code corresponds to only 9 statements (39 instructions, according to EclEmma). These statements refer to exceptions for non-implemented visitors and Velocity errors;
- Code parts for warning parsing errors, typechecking errors or non-compliance of the translation requirements for the input specification (see 4.4.2);

The faults detected on *JCircus*

With the verification strategy applied to the input set described on the previous sub-section, we increased the reliability of ***JCircus*** with respect to the code it generates. During the verification, the JPF oracles detected errors caused by two faults on the implementation of ***JCircus***. The first fault on ***JCircus*** was on the construction of the synchronisation lists for the strategy of Parallelism. ***JCircus*** was overwriting the synchroni-

sation lists when similar invocations were made. For example: for prefixed parallelisms like

process SL $\widehat{=}$ **begin** a $\rightarrow$ (a $\rightarrow$ *Skip* $[\![\{\!|$ a $|\!\}]\!]$ a $\rightarrow$ *Skip*),

the correct branch indexing is:

process SL $\widehat{=}$ **begin** $a^{\{0\}} \rightarrow (a^{\{0\}} \rightarrow$ *Skip* $[\![\{\!|$ a $|\!\}]\!]$ $a^{\{1\}} \rightarrow$ *Skip*),

and the correct synchronisation list is:

$a \rightarrow \{\{0\}, \{0,1\}\}$

As only the prefixing was branched, the sub-expression that came afterwards did not have a branch index:

process SL $\widehat{=}$ **begin** $a^{\{0\}} \rightarrow$ (a $\rightarrow$ *Skip* $[\![\{\!|$ a $|\!\}]\!]$ a $\rightarrow$ *Skip*),

The result of that was that channel *a* had a synchronisation list with only $\{0\}$:

$a \rightarrow \{\{0\}\}$

As channel *a* had only $\{\{0\}\}$ as a synchronisation list, both parallel branches with *a* were marked as the same branch with the same front-end index, and that caused an alting barrier violation: more than one process can not be engaged in the same barrier front-end (see 4.1). When this error was fixed, no other errors were found by the oracles.

The second fault on ***JCircus*** was on the mapping of front-end indexes for events that were performed more than once but with different front-end cardinalities depending on the occurrence. Consider again process SL:

process SL $\widehat{=}$ **begin** a $\rightarrow$ (a $\rightarrow$ *Skip* $[\![\{\!|$ a $|\!\}]\!]$ a $\rightarrow$ *Skip*),

the correct front-end indexing is:

process SL $\widehat{=}$ **begin** $a_{\{0,1\}} \rightarrow (a_{\{0\}} \rightarrow$ *Skip* $[\![\{\!|$ a $|\!\}]\!]$ $a_{\{1\}} \rightarrow$ *Skip*),

The first a has front-end set $\{0,1\}$: it has 2 as cardinality because it is a single occurrence of a prior to a synchronisation between two a's with front-end sets $\{0\}$ and $\{1\}$. So the first occurrence of a is prefixed by the occurrence of the synchronisation between the two a's. ***JCircus*** originally translated the above example as having only front-end set $\{0\}$ on the first a:

process SL $\widehat{=}$ **begin** $a_{\{0\}} \rightarrow (a_{\{0\}} \rightarrow Skip \ [\![\{\!| a |\!\}]\!] \ a_{\{1\}} \rightarrow Skip)$,

The result of that was that the alting barrier of a did not have front-end 1 with a process engaged on it, and, as a result, there was a deadlock, and this contradicted what was expected by the verification: the verification expected two occurrences of a, and no occurrence was originally found.

Although the verification indicated faults on the source code of ***JCircus*** and the correction of such faults increased the reliability of ***JCircus***, there are still limitations on this code generator: not only ***JCircus*** does not cover the whole grammar of *Circus* (see subsection 4.4.2), but also has limitations on the translations of infinite data types. The infinite type $\mathbb{N}$ is translated, but before the translation the user must enter a maximum absolute value for it (the translatable grammar of *Circus* on ***JCircus*** can be seen at appendix A). Moreover, correctness of the code generated by ***JCircus*** is still an open issue.

Scalability of our Strategy

Our strategy has grave scalability problems. Not only the memory consumed by the LPTS explodes for specifications with multiple interleaved branches and multiple events per interleaved branch, but also the memory consumed by the refinement-tree file easily overflows for multiple interleaved branches, even having only one event per interleaved branch. Table 5.2.3 shows data about size and time of generation of the LPTS and size, number of refinement-checking paths and time of refinement-checking for processes with a number of interleaved branches between 2 and 7 branches. Each branch is $a \rightarrow Skip$. Thus, processes with two branches are

$$a \rightarrow Skip \ ||| \ a \rightarrow Skip$$

, with three branches are

$$a \rightarrow Skip \ ||| \ a \rightarrow Skip \ ||| \ a \rightarrow Skip$$

and so on. We use M.O. as an acronym for Memory Overflow:

Num. of Interleaved branches	2	3	4	5	6	7
Num. of Nodes.	8	11	16	21	26	31
Time gen. LPTS (sec)	0.094	0.156	0.241	0.388	0.572	0.747
Time Exec.Ref.Checking (sec)	1	2	7	18	36	-
Num.Paths.Ref.Checking	2	9	24	60	120	-
Size Ref.Tree File (Bytes)	1,961	5,985	25,293	79,260	197,895	M.O.

Data about our strategy for multiple interleaved branches with one event per interleaved branch ($a \rightarrow Skip$)

On table 5.2.3, we can see that 7 branches with 1 event per branch leaded to a Memory Overflow and, as a result, we could not execute the refinement-checking and thus not measure the number of paths of refinement checking and the time of execution. For more events per branch, as it is shown in table 5.2.3, possibility of refinement-checking becomes even more limited: we reached a maximum of 4 interleaved branches for which we managed to generate an LPTS and to perform the refinement-checking.

Num. of Interleaved branches	2	3	4	5
Num. of Nodes.	16	27	47	72
Time gen. LPTS (sec)	0,181	0,428	0.902	1,856
Time Exec.Ref.Checking (sec)	1	4	34	-
Num.Paths.Ref.Checking	2	6	48	-
Size Ref.Tree File (Bytes)	3,805	17,025	177,805	M.O.

Data about our strategy for multiple interleaved branches with 3 events per interleaved branch ($a \rightarrow a \rightarrow a \rightarrow Skip$)

The limitations of performance of our strategy disallowed us to model-check not only case studies but also simple examples, like the Air Controller example (see 3.1). A piece of future work is to optimize the refinement-checking strategy, not only by implementing a Partial Order Reduction (POR) (Peled 1998, Kurshan et al. 1998, Flanagan & Godefroid 2005) on the construction of the Labelled-Predicate Transition System but also implementing an optimized use of memory on the refinement-tree.

Chapter 6

Conclusions and Future Work

We will conclude the PhD thesis and show future work.

6.1 Conclusions

On this PhD thesis, we created and developed a strategy (encoded as a toolchain) to verify the code generation from concurrent and state-rich *Circus* specifications to Java Executable code. The code generation was performed by the translator from *Circus* to Java, ***JCircus***. We also extended the set of rules of Woodcock's Structural Operational Semantics to missing *Circus* actions, in order to strenghten the reliability of the manually constructed tools (LPTS Compiler and JPF Model Generator) based on this semantics (as Woodcock's semantics did not encompass rules for Specification Statements, If-guarded-commands, Hiding Skip, Call Actions, Z Schemas and Iterated Actions) and lifted Woodcock's Structural Operational Semantics to *Circus* processes. This extension was made by the creation of a new kind of transition, the syntactic transition ($\rightsquigarrow$). Then the soundness of the laws of Operational Semantics of *Circus* was proved with respect to the Unifying Theories of Programming (these proofs can be seen on the anexes from appendix C.2 to appendix C.3.21).

From the objectives proposed, we achieved a full and Sound Operational Semantics for *Circus* and a toolchain that, alongside an input set that gave high source code coverage for ***JCircus***, allowed verifying ***JCircus***. The results we achieved with the verification of ***JCircus*** allowed identifying two faults on the source code of ***JCircus***. The correction of these faults increased the reliability of the tool.

One of the difficulties we faced on this work was the lack of tools for model-checking *Circus* specifications. Although Leonardo Freitas (Freitas 2006) had developed an old prototype using different Abstract Syntax Trees from other formalisms similar to *Circus*, it did not fit with the Abstract Syntax Tree of *Circus* that served as a basis to the implementa-

tion of ***JCircus***. Moreover, the prototype did not have a parser. We faced this difficulty developing from scratch a Labelled Predicate Transition System (LPTS) Compiler for *Circus*.

We also created a strategy to verify the refinement between a *Circus* model and a *Circus* generated code (from ***JCircus***), by checking if they are semantically equivalent. This strategy was implemented as a JPF Model Generator, that inputs an LPTS and the generated code by ***JCircus***, and outputs a Java Pathfinder model that, when executed, refinement-checked the generated code by ***JCircus***. One of the difficulties we faced on this development was the explosion of executions when JPF executed constructs from JCSP libraries. This difficulty was overcome by separating the logic of testing acceptance or refusal of events from the part where constructs of JCSP libraries were used, and by executing it natively. As the whole Java code for processes used JCSP libraries, they were entirely executed natively (that is, outside the reach of JPF's virtual Machine). Our solution also encompassed an aspect (in AspectJ) that intercepted the non-deterministic instructions (`Random.nextInt` instructions) on the processes and re-executed them under JPF's Virtual Machine (in order to test all paths of execution).

Each Java Pathfinder Model generated by the toolchain served as a test oracle for its corresponding generated code by ***JCircus***, indicating if the generated code successfully implemented the semantics of the input specification. That is, if the semantics of the input specification was refined by the generated code by ***JCircus***, and vice-versa.

The above strategy was applied to verify ***JCircus***. This verification was made by inputting a set of *Circus* specifications that satisfied high code coverage (Decision Coverage) of the source code of ***JCircus***. The criteria for creating the inputs was guided by a combination of (1) Grammar-based coverage (considering the translatable grammar of *Circus* on appendix A), (2) special/major conditions we established for the source code of ***JCircus***, and (3) manual adjustment of the input set when minor reachable conditions were found on the source code of ***JCircus***.

On the theoretical level, the strategy we developed for checking the semantic equivalence between Model and Code (see formulas 5.1, 5.2, 5.3 and 5.4 on chapter 5) can be easily adapted to any other Process Algebra (being state-rich or not), and is a good starting point for verifying implementations of more complex state-rich Process Algebras, like OhCircus (Cavalcanti et al. 2005) and SCJ-Circus (Miyazawa & Cavalcanti 2016).

6.2 Future Work

Althought there were advancements both on ***JCircus***' development whereas on ***JCircus***' verification, not only Correctness is still an open issue as also there is still work to do: not only (1) the translation strategy from *Circus* to Java still does not encompass the whole grammar of *Circus*, but also (2) the vast majority of the Z part of the grammar of *Circus* does not have a translation strategy. The limitations of JCSP (the library used by the generated code by ***JCircus***) imposed restrictions both on ***JCircus***' translation strategy and on ***JCircus***' performance, and thus, another interesting future work related to ***JCircus*** would be diminishing its number of modules (8 modules), envisaging optimizing its translation from *Circus* to Java. The extension of JCSP in order to support all kinds of CSP constructs would also be an interesting future work.

Our strategy has serious scalability problems. The limitations of Java Pathfinder (JPF) imposed serious restrictions to the performance of our strategy: as JPF exploded the number of interleavings when executed with JCSP, we had to execute the logic of choice generation on the non-native part of JPF and the process on the JPF native part. As a collateral effect, we could not take advantage of JPF's Partial Order Reduction, that could optimize the time of the refinement-checking. An interesting future work would be to investigate a manner to allow JPF's Partial Order Reduction, envisaging a better performance and, thus, a more suitable strategy for bigger *Circus* specifications.

The formal proofs that motivated increasing the reliability of the toolchain were made manually. An interesting future work, envisaging improving the relability of the Soundness of the Operational Semantics of *Circus* would be using an Automated Theorem Prover. There is an embedding of the Unifying Theories of Programming on the Isabelle theorem prover, called Isabelle-UTP (Foster et al. 2014).

On the issue of Correctness, the works quoted by Palsberg about Cantor open a promising direction for constructing a correct compiler for *Circus*: it could be developed an Action Semantics based on the Operational Semantics of *Circus* and use it as input on a Compiler Generator and output a correct compiler for *Circus*.

Appendix A

Translatable Grammar of *Circus* on *JCircus*

The Translatable Grammar of *Circus* on ***JCircus*** is shown using regular BNF notation. Except for lexems **id**, **num** and **|B|**, all other lexems are from tokens of reserved words: these lexems are written like they appear on the grammar. Token **|B|** is a symbol for lexem "|". The acceptable tokens for **id** and **num** are shown by the following regular expressions:

id = ([a-z] | [A-Z])+[0-9]*
num = [0-9]+

CircusSpec $\longrightarrow$ CircusPara CircusSpec | λ
CircusPara $\longrightarrow$ TranslatableZedPara | CircusEnvPara
CircusEnvPara $\longrightarrow$ **begin**{**circus**} CircusEnvParaInside **end**{**circus**}
CircusEnvParaInside $\longrightarrow$ ChannelSetPara | ChannelPara | ProcessPara
ChannelPara $\longrightarrow$ **circchannel** ChannelDef \\
ChannelSetPara $\longrightarrow$ **circchanset id** == TransChanSetExpr \\
TranslatableZedPara $\longrightarrow$
\ **begin**{**zed**} FreetypePara \ **end**{**zed**}
| \ **begin**{**axdef**} AxDefPara \ **end**{**axdef**}
FreetypePara $\longrightarrow$ **id ::=** ElemList
ElemList $\longrightarrow$ **id** | **id |B|** ElemList
AxDefPara $\longrightarrow$ VarDecl | VarDecl **where** Pred
VarDecl $\longrightarrow$ VarList **:** VarTranslatableType
VarTranslatableType $\longrightarrow$ **nat** | **id**
ChannelDef $\longrightarrow$

SimpleTransChannelDef | SimpleTransChannelDef **;** ChannelDef
SimpleTransChannelDef ⟶
id SimpleTransChannelDef
| **id** SimpleTransChannelDef **:** TranslatableChannelExpr | λ
TranslatableChannelExpr ⟶
VarTranslatableType **cross** TranslatableChannelExpr | VarTranslatableType | λ
TransChanSetExpr ⟶
lchanset id Compl
Compl ⟶ **, id** Compl | **rchanset**
ProcessPara ⟶ **circprocess id** **circdef** Proc
Proc ⟶
circbegin ActionParas StateDecl ActionParas **circspot** CircusAction **circend** \\
| **(** Proc **)**
| Decl **circspot** Proc
| Proc **circhide** TransChanSetExpr
| Proc **lcircrename** VarList **:=** VarList **rcircrename**
| **id** | **id (** ExprList **)**
| Proc **extchoice** Proc
| Proc **intchoice** Proc
| Proc **circseq** Proc
| Proc **lpar** TransChanSetExpr **rpar** Proc
| Proc **interleave** Proc
| **Extchoice** VarDecl **circspot** Proc
| **Intchoice** VarDecl **circspot** Proc
| **Semi** VarDecl **circspot** Proc
| **lpar** TransChanSetExpr **rpar** VarDecl **circspot** Proc
| **Interleave** VarDecl **circspot** Proc
ActionParas ⟶ ActionPara ActionParas | λ
StateDecl ⟶ **circstate** StatePara \\ | λ
StatePara ⟶ **id** == [VarDecl | Pred]
ActionPara ⟶ **id** **circdef** CircusAction
CircusAction ⟶ CSPAction | Command | **(** CircusAction **)**
CSPAction ⟶
Skip | **Stop** | **Chaos**
| Comm **then** CircusAction
| **lcircguard** Pred **rcircguard** **circguard** CircusAction

| Decl **circspot** CircusAction
| CircusAction **circhide** TransChanSetExpr
| CircusAction \lcircrename VarList **:=** VarList \rcircrename
| **id** | **id** (ExprList)
| CircusAction **extchoice** CircusAction
| CircusAction **circseq** CircusAction
| CircusAction **intchoice** CircusAction
| CircusAction **interleave** CircusAction
| CircusAction **lpar** TransChanSetExpr **rpar** CircusAction
| CircusAction **lpar** \{ NameSet \} **|B|** TransChanSetExpr **|B|** **rpar** \{ NameSet \} CircusAction
| CircusAction **linter** \{ NameSet \} **rinter** **|B|** **linter** \{ NameSet \} **rinter** CircusAction
| **Interleave** VarDecl **circspot** CircusAction
| **Interleave** VarDecl **linter** NameSet **rinter** **circspot** CircusAction
| **Intchoice** VarDecl **circspot** CircusAction
| **Semi** VarDecl **circspot** CircusAction
| **Parallel** VarDecl **lpar** TransChanSetExpr **rpar** **circspot** CircusAction
Command $\longrightarrow$ **circif** GActions **circfi** | VarList := ExprList | VarDecl **cirspot** CircusAction
VarList $\longrightarrow$ **id** | **id** , VarList
ExprList $\longrightarrow$ Expr | Expr , ExprList
GActions $\longrightarrow$ Pred **circthen** CircusAction | Pred **circthen** CircusAction **circelse** GActions
Comm $\longrightarrow$ **id**Field Comm | λ
Field $\longrightarrow$ **?id** | **!**Expr | **.**Expr
Expr $\longrightarrow$ **(** Expr **)** | Expr BinOpExpr Expr | UnOpExpr Expr | **id**$\backsim$Expr | **num**
BinOpExpr $\longrightarrow$ **+** | **-** | ***** | **/** | **mod**
UnOpExpr $\longrightarrow$ **+** | **-**
Pred $\longrightarrow$ **(** Pred **)** | Pred BOP Pred | UOP Pred | Expr BOPE Expr | **true** | **false**
BOP $\longrightarrow$ **land** | **lor** | **implies** | **iff**
BOPE $\longrightarrow$ **neq** | **=**
UOP $\longrightarrow$ **neg**

Appendix B

Circus Denotational Semantics

On this annex, we will show some used Denotational Semantics laws of *Circus*. All these laws were extracted from (Oliveira 2006*a*).

Definition B.1. ***Variable Block Denotational Definition**:*
var $x : T \bullet A = \exists\, x, x' : T \,.\, A$

Proof The law was created on document (Oliveira 2006*a*).. □

Definition B.2. ***Prefixing Denotational Definition***
$l \rightarrow Skip = R\,(true \vdash doC\,(l, C) \wedge vars' = vars)$

Where

- *l is a communication*
- *doC is defined on C.32*
- *v is the set of variables of the state space (not including either ok, wait, ref or tr)*

The definition was created on (Oliveira 2006*a*).

Definition B.3. ***Skip Denotational Definition**:*
$Skip = R(true \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v)$

The definition was created on (Oliveira 2006*a*).

Definition B.4. ***Assignment Denotational Definition**:*
$x := e = R\,(true \vdash x' = e \wedge u' = u \wedge \neg\, wait')$

Where u = v - $\{x\}$. The definition was created on (Oliveira 2006*a*).

Definition B.5. ***If-Guarded Command Denotational Definition**:*
if $[\!] \; i \bullet g_i \rightarrow A_i$ **fi** $= R\,((\bigvee i \bullet g_i) \wedge (\bigwedge i \bullet g_i \Rightarrow \neg\ A_i{}^{f}_{f}) \vdash \bigvee i \bullet (g_i \wedge A_i{}^{t}_{f}))$

The definition lies on (Oliveira 2006*a*).

Definition B.6. ***Specification Statement Denotational Definition**:*
$w : [pre, post] = R(pre \vdash post \wedge \neg\ wait' \wedge tr' = tr \wedge u' = u)$

The definition lies on (Oliveira 2006*a*).

Appendix C

Proofs of rules of the Operational Semantics of *Circus*

On this appendix we will show the proofs of Soundness of the rules of the Operational Semantics of *Circus* we developed for this thesis. The rules were proved sound with respect to the Denotational Semantics of *Circus*, that is the theory of *Circus* in the Unifying Theories of Programming.

Each proof can be of a lemma, a theorem, a law or a rule from the Operational Semantics of *Circus*. Each proof has a sequence of sub-goals, that are intermediate expressions that compose the proof, and tactics, that transform a sub-goal on another sub-goal. The format of each proof is shown as follows:

Sub-goal 1
Op [Tactic]
...
Op [Tactic]
Sub-goal n

Each tactic is highlighted in **bold**, and each operator **Op** can be either "=", "$\Rightarrow$" or "$\sqsubseteq$". When the tactic uses either a theorem, a lemma, a rule or a definition, it will indicate what theorem, lemma, rule or definition it is and having an hyperlink to it.

C.1 Proving the Soundness of a rule

To prove that a rule of the Operational Semantics of *Circus* is sound with respect to the Denotational Semantics of *Circus* (the theory of *Circus* on the Unifying Theories of Programming) means to prove that the transition from the rule is a predicate that is **true**.

We will exemplify a proof of Soundness for the rule of the Guard construct. The rule for Guard construct was created by (Cavalcanti & Woodcock 2013) and proved correct on this PhD thesis (see table C.1). The rule is shown as follows:

$$\frac{\text{c} \qquad (\text{s} \,;\, \text{g})}{(\text{c} \mid \text{s} \models \text{g} \,\&\, \text{A}) \xrightarrow{\tau} (\text{c} \wedge (\text{s} \,;\, \text{g}) \mid \text{s} \models \text{A})}$$

Thus, what we want is to show that the silent transition

$$(\text{c} \mid \text{s} \models \text{g} \,\&\, \text{A}) \xrightarrow{\tau} (\text{c} \wedge (\text{s} \,;\, \text{g}) \mid \text{s} \models \text{A})$$

is a predicate that is **true**. The factors that lie above the line, *c* and *s*; *g*, are provisos of the proof: a proviso is an expression that is assumed **true** for the proof. So, the rule assumes that *c* is **true** and that *s*; *g* is **true**.

Before proving the expression above, we will create a theorem and prove it: it is called Refinement Equal Sides:

Theorem - Refinement Equal Sides: EXPR $\sqsubseteq$ EXPR

Proof:

EXPR $\sqsubseteq$ EXPR

(The expression we want to prove is the one above. A refinement expression
E1 $\sqsubseteq$ E2 equals [E2 $\Rightarrow$ E1], where the brackets mean universal quantification on the free-variables of the expression:)
= [EXPR $\Rightarrow$ EXPR]

(Now we apply a Predicate Calculus: an implication expression consists on a disjunction ("or" operator) between the negation of the left side and the right side:)
= [$\neg$ EXPR $\vee$ EXPR]
= [true]
= true

We start our proof with the first sub-goal, which is the expression itself:

Proof:

$(c \mid s \models g \;\&\; A) \xrightarrow{\tau} (c \wedge (s \,;\, g) \mid s \models A)$

(As we saw from the definitions of Silent and Labelled Transition (see 3.1 and 3.2), a transition on the Operational Semantics is, in fact, a predicate expression on the UTP. So, the first tactic that we will apply on the proof is to transform the operational representation of the transition into its denotational predicate expression, resulting on the predicate shown below)
$= \forall\, w \,.\, (c \wedge c \wedge (s \,;\, g)) \Rightarrow ((\text{Lift}\ (s) \,;\, g \;\&\; A) \sqsubseteq (\text{Lift}\ (s) \,;\, A))$

(Our second tactic deals with the presence of variable w being universally quantified. When we prove that a given predicate is true, we prove that it is true for all possible values of its variables. So, during a proof, the whole expression is universally quantified, including w. So we can abstract the occurrence of w:)
$= (c \wedge c \wedge (s \,;\, g)) \Rightarrow ((\text{Lift}\ (s) \,;\, g \;\&\; A) \sqsubseteq (\text{Lift}\ (s) \,;\, A))$

(We have, as an assumption, that c is true. So, our tactic now is to replace all occurrences of c by true)
$= (\text{true} \wedge \text{true} \wedge (s \,;\, g)) \Rightarrow ((\text{Lift}\ (s) \,;\, g \;\&\; A) \sqsubseteq (\text{Lift}\ (s) \,;\, A))$

(Our tactic now is to apply a predicate calculus such that true $\wedge$ P equals P)
$= (\text{true} \wedge (s \,;\, g)) \Rightarrow ((\text{Lift}\ (s) \,;\, g \;\&\; A) \sqsubseteq (\text{Lift}\ (s) \,;\, A))$

(We apply, again, the same tactic: true $\wedge$ P equals P)
$= ((s \,;\, g)) \Rightarrow ((\text{Lift}\ (s) \,;\, g \;\&\; A) \sqsubseteq (\text{Lift}\ (s) \,;\, A))$

*(We now divide the proof in two parts: one in which we prove the above expression is true for g being **true**, and the other in which we prove that the above expression is **true** for g being false. If we prove that, for both cases, the above expression is true, then it is true for any value of g)*

Part 1 - for g = true:

$(s \,;\, g) \Rightarrow ((\text{Lift}\ (s) \,;\, g \;\&\; A) \sqsubseteq (\text{Lift}\ (s) \,;\, A)))$

(We simply replace g by true)
= (s ; true) ⇒ ((Lift (s) ; true & A ⊑ (Lift (s) ; A))

*(This tactic now transforms expression true & A into A using a theorem called True Guard, that says that g & A = A. This theorem lies on (Oliveira 2006*a*))*
= (s ; true) ⇒ ((Lift (s) ; A) ⊑ (Lift (s) ; A)

(Now we apply the theorem we have proved correct: Refinement Equal Sides)
= (s ; true) ⇒ true

(We apply now Predicate Calculus: when the consequent of an implication is true, then the whole expression is true:)
= **true**

Part 2 - for g = false:

(s ; g) ⇒ ((Lift (s) ; g & A) ⊑ (Lift (s) ; A)))

(We simply replace g by false)
= (s ; false) ⇒ ((Lift (s) ; false & A) ⊑ (Lift (s) ; A))

(We remove w from the expression using a similar strategy from Part 1:)
= (s ; false) ⇒ ((Lift (s) ; false & A) ⊑ (Lift (s) ; A))

(There is a theorem, on (Hoare & Jifeng 1998), that says that s; *false equals false. So we make the replacement:)*
= false ⇒ ((Lift (s) ; false & A) ⊑ (Lift (s) ; A))

(By Predicate Calculus, if we have false on the antecedent of an implication, then the whole expression is true)
= true

The proof for rule of Guard is shown at 11. Beyond the proofs of soundness that we have made for the rules of *Circus* (some of which created by ourselves), we also encompass, on this document, previous results on the soundness of the Operational Semantics of Woodcock's Operational Semantics. We will show a summary of rules.

Rule (CML/Circus Actions)	Created by	Proved by
Assignment	(Woodcock et al 2013)	(Woodcock et al 2013)
Input (Prefixing)	(Woodcock et al 2013)	(Woodcock et al 2013)
Output (Prefixing)	(Woodcock et al 2013)	(Woodcock et al 2013)
Variable Block Begin	(Woodcock et al 2013)	Samuel Barrocas
Variable Block Visible	(Woodcock et al 2013)	Samuel Barrocas
Variable Block End	(Woodcock et al 2013)	Samuel Barrocas
Sequence Progress	(Woodcock et al 2013)	Samuel Barrocas
Sequence End	(Woodcock et al 2013)	Samuel Barrocas
Internal Choice Left	(Woodcock et al 2013)	(Woodcock et al 2013)
Internal Choice Right	(Woodcock et al 2013)	(Woodcock et al 2013)
Guard	(Woodcock et al 2013)	Samuel Barrocas
External Choice Begin	(Woodcock et al 2013)	(Woodcock et al 2013)
External Choice End	(Woodcock et al 2013)	(Woodcock et al 2013)
External Choice Skip	(Woodcock et al 2013)	(Woodcock et al 2013)
Ext. Choice Silent Left	(Woodcock et al 2013)	Samuel Barrocas
Ext. Choice Silent Right	(Woodcock et al 2013)	Samuel Barrocas
Parallel Begin	(Woodcock et al 2013)	(Woodcock et al 2013)
[1]Parallel Ind. Left (PIL)	(Woodcock et al 2013)	Samuel Barrocas
[2]Parallel Indep. Right	(Woodcock et al 2013)	Samuel Barrocas
Parallel Synchronised	(Woodcock et al 2013)	Samuel Barrocas
Parallel End	(Woodcock et al 2013)	(Woodcock et al 2013)
Hiding Internal	(Woodcock et al 2013)	Samuel Barrocas
Hiding Visible	(Woodcock et al 2013)	Samuel Barrocas
Hiding Skip	Samuel Barrocas	Samuel Barrocas
Recursion	(Woodcock et al 2013)	Samuel Barrocas

Rule (CML/Circus Actions)	Created by	Proved by
If-Guarded-Command	Samuel Barrocas	Samuel Barrocas
Call Action No Params	Samuel Barrocas	Samuel Barrocas
Call Action W/Params	Samuel Barrocas	Samuel Barrocas
Rename Action	Samuel Barrocas	Samuel Barrocas
Alphabetised Par. Act.	Samuel Barrocas	Not proved (future work)
Parameter Action Call	Samuel Barrocas	Samuel Barrocas
Iterated Actions	Samuel Barrocas	Samuel Barrocas
Parameter Action	Samuel Barrocas	Samuel Barrocas
Specification Statement	Samuel Barrocas	Samuel Barrocas
Schema Action	Samuel Barrocas[3]	Samuel Barrocas

Circus process	Created by	Proved by
Basic Process Begin	Samuel Barrocas[4]	Samuel Barrocas
Basic Process Reduction	Samuel Barrocas	Samuel Barrocas
Hiding Advance	Samuel Barrocas	Samuel Barrocas
Hiding Basic Process	Samuel Barrocas	Samuel Barrocas
Rename Advance	Samuel Barrocas	Samuel Barrocas
Rename Basic Process	Samuel Barrocas	Samuel Barrocas
Compound Process Left	Samuel Barrocas	Samuel Barrocas
Compound Process Right	Samuel Barrocas	Samuel Barrocas
Call process with normal parameters	Samuel Barrocas	Samuel Barrocas
Call process with indexed parameters	Samuel Barrocas	Samuel Barrocas
Call process with generic parameters	Samuel Barrocas	Samuel Barrocas
Iterated Processes	Samuel Barrocas	Samuel Barrocas

[1] and [2]: in the case of Parallel Independent Left and Parallel Independent Right, (Woodcock et al 2013) created a section to prove the soundness of these rules, but did not prove it; [3]: this rule was adapted from (Cavalcanti & Gaudel 2011) and proved correct on this PhD thesis;

C.2 Auxiliary Definitions, Lemmas, Theorems and Laws

This appendix shows all definitions, lemmas, theorems and laws used on (or created by) this document. Those marked with a * consist on definitions that were created or quoted by other authors (quoted after each definition). Those without * were created on this document. We will begin showing Auxiliary Definitions.

C.2.1 Auxiliary Definitions

Definition C.1. ***getAssignments*** *is an auxiliary function (created for this document) that calculates the sequence of assignments of a node whose program text is a process.*
When the parameter of getAssignments is a Circus action Act, it returns the assignments s of the node whose program text is Act:

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{a} (c_2 \mid s_2 \models A_2)}{getAssignments\ (A_1) = s_1 \land getAssignments\ (A_2) = s_2}$$

For Example:

$(\textbf{true} \mid \{\ \} \models a \rightarrow STOP) \xrightarrow{a} (\textbf{true} \mid \{\ \} \models STOP)$
$getAssignments\ (a \rightarrow STOP) = \{\ \}$

When the parameter is a Circus process, getAssignments has the following definition:

- *getAssignments (***begin state** *ST = [decl | inv] • Act* **end***) = getAssignments (Act)*
- *getAssignments (Call) = getAssignments (Content(Call))*
- *getAssignments (P_1 **bop** P_2) = getAssignments (P_1) ∧ getAssignments (P_2), where **bop** is a binary operator (either an external choice, internal choice, interleaving, parallelism and etc)*
- *getAssignments ($P \setminus CS$) = getAssignments (P)*
- *getAssignments (P [RenamingPairs]) = getAssignments (P)*
- *getAssignments (ITOP Decl • P) = getAssignments (IteratedExpansion (P, Decl, ITOP))*
- *getAssignments (P [N+]) = getAssignments (GenericContent (P, [N+]))*

Definition C.2. ***pre*** *is an unary function that calculates the precondition of a given Z Schema. This calculation is performed by guaranteeing that there is a future state and by hiding the output variables of the schema (those that have a "!" after their names on the definition of the schema):*

pre *Schema* = $\exists$ *State' . Schema* $\setminus$ *Outputs*

This definition lies on (Woodcock & Davies 1996).

Definition C.3. ***IteratedExpansion*** *is a function that unfolds an iterated operator applied to an action or to a process. It has three parameters: the first parameter is the process itself, the second is the declaration of the variables used to iterate the process (or the action), and the third is a flag indicating what is the operator applied to the iteration. The type of the iterating variables must be finite, or infinite with range previously specified. For Example:*

IteratedExpansion (a.x $\rightarrow$ *Skip, x :* $\{0, 1, 2, 3\}$*, EXTCHOICE) =*
a.0 $\rightarrow$ *Skip* $\Box$ *a.1* $\rightarrow$ *Skip* $\Box$ *a.2* $\rightarrow$ *Skip* $\Box$ *a.3* $\rightarrow$ *Skip*

Definition C.4. ***Body*** *is a function that unfolds the body of a call action. For Example:*

process $P \mathrel{\widehat{=}}$ **begin**
$A \mathrel{\widehat{=}} a \rightarrow Skip$
$\bullet$ A
end

Body (A) = a $\rightarrow$ *Skip.*

Body *can also be used for processes. For Example:*

Body (P) =

begin
$A \mathrel{\widehat{=}} a \rightarrow Skip$
$\bullet$ A
end

Definition C.5. ***IndexedContent*** *works as* ***Body****, but is applied to processes with indexed parameters.*
process $P \mathrel{\widehat{=}}$ **begin** • *request?x* $\rightarrow$ *response!x* $\rightarrow$ *Skip* **end**
process $Q \mathrel{\widehat{=}} x : \{0, 1\} \odot P$
IndexedContent (Q, [0]) = $Q \lfloor 0 \rfloor$
= **begin** • *request_0.0?x* $\rightarrow$ *response_0.0!0* $\rightarrow$ *Skip* **end**

Definition C.6. ***ParamContent*** *works as* ***Body****, but is applied to processes with normal parameters. For Example:*

process $P \mathrel{\widehat{=}} x : \mathbb{N};\ y : \mathbb{N}$ • **begin**
• *a.x.y* $\rightarrow$ *Skip*
end

ParamContent (P, [1, 2]) = **begin** • *a.1.2* $\rightarrow$ *Skip* **end**

Definition C.7. ***GenericContent*** *works as* ***Content****, but is applied to processes with generic parameters. GenericContent (P, [N+]) takes process P and a list of generic types [N+] as parameters. For Example:*

channel *[C] singleevent : C*
process *P [T]* $\mathrel{\widehat{=}}$ *begin* • *singleevent [T]?x* $\rightarrow$ *Skip* **end**
process *Inst* $\mathrel{\widehat{=}}$ *P* [$\mathbb{N}$]

On the specification above,

GenericContent (P, [$\mathbb{N}$]) =
begin • *singleevent* [$\mathbb{N}$]*?x* $\rightarrow$ *Skip* **end**

Definition C.8. *A = Body (A), provided that A is a call.*

Definition C.8 says that any call action A equals its body. Given:

process $P \mathrel{\widehat=}$ **begin**
$A \mathrel{\widehat=} a \rightarrow Skip$
$\bullet\ A$
end

The call action A equals a → Skip, which is Body (A) (def. C.4). Thus, A = Body (A).

Definition C.9. *A (N+) = ParamContent (A, [N+]), provided that A is a parameterised call.*

Definition C.9 says that any parameterised call action A equals its body having the parameters replaced by their call expressions. Given:

process $P \mathrel{\widehat=}$ **begin**
$A \mathrel{\widehat=} x : \mathbb{N} \bullet a.x \rightarrow Skip$
$\bullet\ A\,(0)$
end

The call action A (0) equals a.0 → Skip, which is ParamContent (A, [0]) (def. C.6). Thus, A (0) = ParamContent (A, [0]).

Definition C.10. *A ⌊ N+ ⌋ = IndexedContent (A, [N+]), provided that A is an indexed call. Given:*

process $P \mathrel{\widehat=}$ **begin**
$A \mathrel{\widehat=} x : \mathbb{N} \bullet a.x \rightarrow Skip$
$\bullet\ A \lfloor 0 \rfloor$
end

The indexed call action A ⌊ 0 ⌋ equals a_0.0 → Skip, which is IndexedContent (A, [0]) (def. C.5). Thus, A ⌊ 0 ⌋ = IndexedContent (A, [0]).

Definition C.11. *P [N+] = GenericContent (P, [N+]), provided that P is a call. Given the example on C.7,*

$P\ [\ \mathbb{N}\ ] =$ **begin** $\bullet$ *singleevent* $[\ \mathbb{N}\ ]?x \rightarrow Skip$ **end**,

which is GenericContent (P, [ℕ]). Thus, P [ℕ] equals GenericContent (P, [ℕ]).

Definition C.12. ***UsedC** is a function that gets a Circus action as parameter and returns a set of channels that are used within that action. For Example:*
UsedC $(\mu\ X \bullet (turnOn \rightarrow X) \sqcap (turnOff \rightarrow X)) = \{turnOn, turnOff\}$

This definition was created on (Oliveira 2006*a*).

Definition C.13. ***initials** is a function that gets a Circus action as parameter and returns a set of events that are initially offered by the action. For Example:*
initials $(turnOn \rightarrow Skip) = \{turnOn\}$
initials $(turnOn \rightarrow Skip \ \Box\ turnOff \rightarrow Skip) = \{turnOn, turnOff\}$

This definition was created on (Oliveira 2006*a*).

Definition C.14. ***Refinement Definition:***
$P \sqsubseteq Q = [Q \Rightarrow P] = \forall\ v : FREEV\ .\ Q \Rightarrow P$

On the expression above, the FREEV set represents the set of free (non-quantified) variables on the expression between brackets. The brackets mean universal quantification on the free-variables.

The definition above lies on (Hoare & Jifeng 1998, Oliveira 2006*a*)

Definition C.15. ***Reactive Skip***
$II_{rea} = (\neg\ ok \wedge tr \leq tr') \vee (ok' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v)$

This definition was quoted from (Oliveira 2006*a*).

Definition C.16. ***State Assignment***

A Configuration State assignment is defined on (Cavalcanti & Woodcock 2013) as a total function that maps the programming variables in the alphabet to (possibly loose) constants. That leads us to the formula

$(x := c) = (x' = c \land v' = v)$,

where c is a (possibly loose) constant (the value of c is determined as an expression that appears on the constraint of the configuration). The v' = v factor (v is the set of all non-used variables by the assignment) comes from the fact that the function is total, and all the variables on the state space that were not used by the assignment (including the observational variables ok, tr, wait and ref) must have their values mapped to their current values. A consequence of this description is that another possible definition is

$(x := c) = x' = c \land wait' = wait \land ok' = ok \land tr' = tr \land ref' = ref \land nalocalvars' = nalocalvars$,

Where nalocalvars is the set of all program (non-observational) variables (except x, because x lies on the assignment) of the state space. We can generalise the definition above for any assignment s using a ***factor f*** *such that, for example, if*

$s = (x, y := 0, 1)$

then $\mathbf{f} = \mathbf{f}(\alpha, \alpha') = (x' = 0 \land y' = 1)$.

So, for any Configuration State Assignment s,

$s = s(\alpha, \alpha') = f \land wait' = wait \land ok' = ok \land tr' = tr \land ref' = ref \land nalocalvars' = nalocalvars$.

Definition C.17. ***Reactive Design Assignment***

$v :=_{RD} e = Lift(v := e) ; Skip$

This definition was created on (Cavalcanti & Woodcock 2013).

Definition C.18. ***Conditional***

$P \lhd b \rhd Q = (b \wedge P) \vee (\neg\, b \wedge Q)$

This definition was quoted from document (Woodcock & Cavalcanti 2004).

Definition C.19. ***R1***

$R1\ (P) = P \wedge tr \leq tr'$

This definition lies on documents (Cavalcanti & Woodcock 2013, Oliveira 2006*a*).

Definition C.20. ***R2***

$R2\ (P\ (tr, tr')) = P\ (\langle\rangle, tr' - tr) = P\ [\langle\rangle, (tr' - tr) / tr\ , tr']$

This definition lies on document (Oliveira 2006*a*)..

Definition C.21. ***R3***

$R3\ (P) = II_{rea} \lhd wait \rhd P$

This definition lies on documents (Cavalcanti & Woodcock 2013, Oliveira 2006*a*). The Conditional operator is defined on C.18 and II_{rea} (Reactive Skip) is defined on C.15.

Definition C.22. ***R***

$R\ (P) = R1 \circ R2 \circ R3\ (P)$

This definition lies on document (Oliveira 2006*a*)..

Definition C.23. ***Design***

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

This definition lies on document (Oliveira 2006*a*, Woodcock & Cavalcanti 2004).

Definition C.24. ***Lift***

$Lift\ (s) = R1 \circ R3\ (true \vdash s \wedge tr' = tr \wedge \neg\ wait')$

This definition is original from the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for *Circus* (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013).

Definition C.25. ***Silent Transition**:*
$(c_1 \mid s_1 \models A_1) \xrightarrow{\tau} (c_2 \mid s_2 \models A_2)$
=
$\forall\, w \,.\, c_1 \wedge c_2 \Rightarrow \mathit{Lift}\ (s_1)\ ;\ A_1 \sqsubseteq \mathit{Lift}\ (s_2)\ ;\ A_2$

This definition is original from the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for *Circus* (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013).

Definition C.26. ***Labelled Transition**:*
$(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)$
=
$\forall\, w \,.\, c_1 \wedge c_2 \Rightarrow \mathit{Lift}\ (s_1)\ ;\ A_1 \sqsubseteq (\mathit{Lift}\ (s_2)\ ;\ l \rightarrow A_2) \,\Box\, (\mathit{Lift}\ (s_1)\ ;\ A_1)$
This definition is original from the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for Circus (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013).

Definition C.27. ***Syntactic Transition (for processes)****:*

$$(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2)$$
$$=$$
$$\forall\, w \,.\, c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} (\ Lift\,(getAssignments(P_1))\,;\ P_1 \sqsubseteq Lift\,(getAssignments\,(P_2))\,;\ P_2\ ) \\ \wedge\ Lift\,(getAssignments\,(P_1)) = Lift\,(getAssignments\,(P_2)) \end{array} \right)$$

$$(c_1 \models P_1) \rightsquigarrow (c_2 \mid s \models BasicProc)$$
$$=$$
$$\forall\, w \,.\, c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift(getAssignments(P_1))\,;\ P_1 \sqsubseteq Lift(s)\,;\ BasicProc)) \\ \quad \wedge\ Lift(getAssignments(P_1)) = Lift(s) \end{array} \right)$$

The definition of the getAssignments function is on C.1.

The definition of Syntactic Transition C.27 was created on this document.

Definitions C.25, C.26 and C.27 define advances on the program. These advances are refinements between *Circus* denotational expressions: when these refinement expressions are proved correct with respect to the theory of *Circus* on the Unifying Theories of Programming (UTP), it means that the advances of the program whose definitions are these refinement expressions are correct and satisfy the theory.

Definition C.28. ****let*** *(Cavalcanti & Woodcock 2013)*
let $x \bullet A = A$

This definition is original from the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for *Circus* (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013).

Definition C.29. ****loc*** *(Cavalcanti & Woodcock 2013)*
loc $s \bullet P = Lift\ (s)\ ;\ P$

This definition is original from the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for *Circus* (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013).

Definition C.30. ***Extra Choice***
$A_1 \boxplus A_2 = A_1 \Box A_2$

This definition is original from the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for *Circus* (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013). The Extra Choice symbol ($\boxplus$) appears on the rules of External Choice on the Operational Semantics of (Cavalcanti & Woodcock 2013) as "[+]".

Definition C.31. ***Extra Statement***
V : [Pre, Post] = V [:] [Pre, Post]

This definition was created on this thesis. The role of this definition is to represent the intermediate state between the initial state of the Specification Statement (the one in which the Program Text equals V : [Pre, Post]) and an intermediate state where precondition Pre is a factor on the constraint of the final state (the one in which the Program Text equals V [:] [Pre, Post]). This behaviour lies on Attached Rule 30.

Definition C.32. ***doC***
$doC\ (l, C) = (tr' = tr \land (l, C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l, C) \rangle)$

Where

- *l is a channel;*
- *C is a communication;*
- $\langle (l, C) \rangle$ *is a trace that is concatenated to tr;*

This function is used on the Denotational definition of the prefixing operator.

The definition was created on (Oliveira 2006*a*).

Definition C.33. $A\ ;\ B = \exists v0 \ .\ A\ [v, v0] \land B\ [v0, v']$

The above definition was made on (Hoare & Jifeng 1998) and is referenced on (Oliveira 2006*a*).

Definition C.34. ***UTP Variable Declaration***
$(var\ x);\ P = \exists\ x\ .\ P$
The definition was created on (Woodcock & Cavalcanti 2004).

Definition C.35. ***Substitution**:*
$A\,{}^{b}_{c} = A\ [b\ /\ ok'][c\ /\ wait]$

The definition lies on (Oliveira 2006*a*). It is referenced on the denotational definition of other constructs of *Circus*, like If-Guarded Command B.5. When Substitution is referenced, it has factors as $A\,{}^{f}_{f}$ and $A\,{}^{t}_{f}$. On these cases, f means **false** and t means **true**.

Definition C.36. ***Disjunction Reactive Designs**:*
$R\ (P1 \vdash Q1) \lor R\ (P2 \vdash Q2) = R\ (P1 \land P2 \vdash P1 \lor P2)$

The definition lies on (Oliveira 2006*b*).

Definition C.37. ***External Choice Denotational Definition**:*
$A_1 \Box A_2 = R\ ((\neg\ A_1\,{}^{f}_{f} \land \neg\ A_2\,{}^{f}_{f}) \vdash ((A_1\,{}^{t}_{f} \land A_2\,{}^{t}_{f}) \lhd tr' = tr \land wait' \rhd (A_1\,{}^{t}_{f} \lor A_2\,{}^{t}_{f})))$

The definition lies on (Oliveira 2006*a*).

Definition C.38. ***CSP1**:*
$CSP1\ (P) = P \lor (\neg\ ok \land tr \leq tr')$

The definition lies on (Oliveira 2006*a*).

Definition C.39. ***CSP2**:*
$CSP2\ (P) = P\ ;\ ((ok \Rightarrow ok') \land tr' = tr \land wait' = wait \land ref' = ref \land v' = v)$

The definition lies on (Oliveira 2006*a*).

Definition C.40. ***CSP3**:*
$CSP3\ (P) = Skip\ ;\ P$

The definition lies on (Oliveira 2006*a*).

C.2.2 Auxiliary Lemmas

Lemma C.1. ***Refinement Healthiness Conditions***
$(II_{rea} \lhd wait \rhd X)\ (\langle\rangle, tr'\ -\ tr)$
$\sqsubseteq (II_{rea} \lhd wait \rhd false)\ (\langle\rangle, tr'\ -\ tr)$
Proof
For wait = true
$((II_{rea})\ (\langle\rangle, tr'\ -\ tr)$
$\sqsubseteq (II_{rea})\ (\langle\rangle, tr'\ -\ tr))$
= [Refinement Equal Sides 1]

true
For wait = false
X (⟨⟩, tr' - tr)
⊑ (false) (⟨⟩, tr' - tr)
= [Predicate Calculus]
X (⟨⟩, tr' - tr) ⊑ false
= [Refinement Definition C.14 and Predicate Calculus]
true □

Lemma C.2. ***Refinement Healthiness Conditions with R1***

$$R(X) \sqsubseteq R(false)$$

Proof:

$R(X) \sqsubseteq R(false)$
$=$ ***[Definitions of R1 C.19, R2 C.20 and R3 C.21]***
$(tr \leq tr' \wedge (II_{rea} \lhd wait \rhd X)(\langle\rangle, tr' - tr))$
$\sqsubseteq (tr \leq tr' \wedge (II_{rea} \lhd wait \rhd false)(\langle\rangle, tr' - tr))$
$=$ ***[Refinement Healthiness Conditions C.1 and Refinement Monotonic 6]***
true

Lemma C.3. ***Expanded Reactive Design Refined by Reactive Ok Negation***
$(R((\neg Ap \wedge ok) \Rightarrow (Ap \wedge ok')) \sqsubseteq R(\neg ok))$

Proof:

For $ok = true$:

$(R((\neg Ap \wedge true) \Rightarrow (Ap \wedge ok')) \sqsubseteq R(\neg true))$
[Predicate Calculus]
$(R((\neg Ap \wedge true) \Rightarrow (Ap \wedge ok')) \sqsubseteq R(false))$
[Refinement Healthiness Condition with R1 C.2]
true

For $ok = false$:

$(R((\neg\ Ap \wedge false) \Rightarrow (Ap \wedge ok')) \sqsubseteq R(\neg\ false))$
$=$ ***[Predicate Calculus]***
$(R(false \Rightarrow (Ap \wedge ok')) \sqsubseteq R(true))$
$=$ ***[Predicate Calculus]***
$(R(true) \sqsubseteq R(true))$
$=$ ***[Refinement Equal Sides 1]***
$true$

Lemma C.4. ***Refinement Implication For Substitution Equivalence*** $(A \sqsubseteq A \ \Box\ B) \Rightarrow (A \sqsubseteq B)$,

provided that $A^{\,f}_{\,f} = A^{\,t}_{\,f}$ *and* $B^{\,f}_{\,f} = B^{\,t}_{\,f}$

Proof:

$Be\,Ap\ =\ Ap^{\,f}_{\,f} = Ap^{\,t}_{\,f}, and\, Bp\ =\ Bp^{\,f}_{\,f} = Bp^{\,t}_{\,f}$

$(A \sqsubseteq A \ \Box\ B) \Rightarrow (A \sqsubseteq B)$
$=$ ***[External Choice Denotational Definition C.37 + Assumption]***
$(R(\neg\ Ap \vdash Ap) \sqsubseteq R((\neg\ Ap \wedge \neg\ Bp) \vdash ((Ap \wedge Bp) \lhd tr' = tr \wedge wait' \rhd (Ap \vee Bp))))$
$\Rightarrow$
$(R(\neg\ Ap \vdash Ap) \sqsubseteq R(\neg\ Bp \vdash Bp))$

$For\,Bp = true$:

$(R(\neg\ Ap \vdash Ap) \sqsubseteq R((\neg\ Ap \wedge \neg\ true) \vdash ((Ap \wedge true) \lhd tr' = tr \wedge wait' \rhd (Ap \vee true))))$
$\Rightarrow$
$(R(\neg\ Ap \vdash Ap) \sqsubseteq R(\neg\ true \vdash true))$
$=$ ***[Predicate Calculus]***
$(R(\neg\ Ap \vdash Ap) \sqsubseteq R(false \vdash (Ap \lhd tr' = tr \wedge wait' \rhd true)))$
$\Rightarrow$
$(R(\neg\ Ap \vdash Ap) \sqsubseteq R(false \vdash true))$
$=$ ***[Predicate Calculus]***
$(R(\neg\ Ap \vdash Ap) \sqsubseteq R(false \vdash (Ap \lhd tr' = tr \wedge wait' \rhd true)))$
$\Rightarrow$

$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(false))$
$=$ ***[Refinement Healthiness Condition with R1 C.2]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(false \vdash (Ap \lhd tr' = tr \wedge wait' \rhd true)))$
$\Rightarrow$
$true$
$=$ ***[Predicate Calculus]***
true

$For\, Bp = false$:

$(R(\neg\, Ap \vdash Ap) \sqsubseteq R((\neg\, Ap \wedge \neg\, false) \vdash ((Ap \wedge false) \lhd tr' = tr \wedge wait' \rhd (Ap \vee false))))$
$\Rightarrow$
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(\neg\, false \vdash false))$
$=$ ***[Predicate Calculus]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R((\neg\, Ap \wedge \neg\, false) \vdash ((Ap \wedge false) \lhd tr' = tr \wedge wait' \rhd (Ap \vee false))))$
$\Rightarrow$
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(\neg\, false \vdash false))$
$=$ ***[Predicate Calculus]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R((\neg\, Ap \wedge true) \vdash ((Ap \wedge false) \lhd tr' = tr \wedge wait' \rhd (Ap \vee false))))$
$\Rightarrow$
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(true \vdash false))$
$=$ ***[Predicate Calculus]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(\neg\, Ap \vdash (false \lhd tr' = tr \wedge wait' \rhd Ap)))$
$\Rightarrow$
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(true \vdash false))$
$=$ ***[Definition of Conditional C.18 + Predicate Calculus]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(\neg\, Ap \vdash (\neg\, (tr' = tr \wedge wait') \wedge Ap)))$
$\Rightarrow$
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(true \vdash false))$
$=$ ***[Design Definition C.23]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(\neg\, Ap \vdash (\neg\, (tr' = tr \wedge wait') \wedge Ap)))$
$\Rightarrow$
$(R((\neg\, Ap \wedge ok) \Rightarrow (Ap \wedge ok')) \sqsubseteq R(\neg\, ok))$
$=$ ***[Expanded Reactive Design Refined by Reactive Ok Negation C.3]***
$(R(\neg\, Ap \vdash Ap) \sqsubseteq R(\neg\, Ap \vdash (\neg\, (tr' = tr \wedge wait') \wedge Ap)))$
$\Rightarrow$

$true$
$=$ ***[Predicate Calculus]***
$true$

Lemma C.5. ***Lift Substitution Equivalence***
$(Lift(s))^{\,f}_{f} = (Lift(s))^{\,t}_{f}$

Proof:

$(Lift(s))^{\,f}_{f}$
$=$ ***[Definition of Lift C.24]***
$(R1 \circ R3(true \vdash s \wedge tr' = tr \wedge \neg\, wait'))^{\,f}_{f}$
$=$ ***[Definitions of R1 C.19 and R3 C.21]***
$(tr \leq tr' \wedge (II_{rea} \lhd wait \rhd (true \vdash s \wedge tr' = tr \wedge \neg\, wait')))^{\,f}_{f}$
$=$ ***[Substitution C.35]***
$tr \leq tr' \wedge (II_{rea}{}^{\,f}_{f} \lhd false \rhd (true \vdash s \wedge tr' = tr \wedge \neg\, wait'))$
$=$ ***[Definition of Conditional C.18 and Predicate Calculus]***
$tr \leq tr' \wedge (true \vdash s \wedge tr' = tr \wedge \neg\, wait')$
$=$ ***[Definition of Conditional C.18 and Predicate Calculus]***
$tr \leq tr' \wedge (II_{rea}{}^{\,t}_{f} \lhd false \rhd (true \vdash s \wedge tr' = tr \wedge \neg\, wait'))$
$=$ ***[Substitution C.35]***
$tr \leq tr' \wedge (II_{rea} \lhd wait \rhd (true \vdash s \wedge tr' = tr \wedge \neg\, wait'))^{\,t}_{f}$
$=$ ***[Definitions of R1 C.19 and R3 C.21]***
$(R1 \circ R3(true \vdash s \wedge tr' = tr \wedge \neg\, wait'))^{\,t}_{f}$
$=$ ***[Definition of Lift C.24]***
$(Lift(s))^{\,t}_{f}$

Lemma C.6. ***Healthy True Refinement:*** $\forall\, x \,.\, R\,(true) \sqsubseteq R(x)$

Proof
$\forall\, x \,.\, R\,(true) \sqsubseteq R(x)$
= [Quantifier $\forall$ ***x can be omitted]***
$R\,(true) \sqsubseteq R(x)$

For x = true:
$(R\,(true) \sqsubseteq R(true))$

*= **[Refinement Equal Sides 1]***
true

For x = false:
(R (true) ⊑ R(false))
*= **[Definitions of R1 C.19, R2 C.20 and R3 C.21]***
tr ≤ tr' ∧ (II_{rea} ◁ wait ▷ true) (⟨⟩, tr' - tr)
⊑ tr ≤ tr' ∧ (II_{rea} ◁ wait ▷ false) (⟨⟩, tr' - tr)
*= **[Ref. Conjunctive Monotonic C.13 and Ref. Healthiness Conditions C.1]***
true

Lemma C.7. ***Design of Assignment R2:***
(true ⊢ x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait') is R2

Proof
true ⊢ x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait'
= [Expression does not contain tr' neither tr]
true ⊢ x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait' [⟨⟩ , tr' - tr / tr , tr']
= [Predicate Calculus]
true ⊢ x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait' (⟨⟩, tr' - tr)
= [Definition of R2 C.20]
R2 (true ⊢ x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait')
□

Lemma C.8. *(Lift (var x ; x := w_0) ; A) ⇒ (**var** x : T • A),*

provided that

- *w_0 : T*

Proof
(Lift (var x ; x := w_0) ; A)
*= **[Lift CSP4 C.2.26]***
(Lift (var x ; x := w_0) ; Skip ; A)
*= **[Definition of Lift C.24]***
(R1 ∘ R3 (true ⊢ var x ; x := w_0 ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)
*= **[Design of Assignment R2 C.7]***
(R1 ∘ R3 ∘ R2 (true ⊢ var x ; x := w_0 ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [R2 and R3 Composition Commutative C.2.49]

(R1 ∘ R2 ∘ R3 (true ⊢ var x ; x := w_0 ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [Definition of R C.22]

(R (true ⊢ var x ; x := w_0 ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [State Assignment C.16]

(R (true ⊢ var x ; x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [UTP Variable Declaration C.34]

(R (true ⊢ ∃ x . x' = w_0 ∧ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [Assumption w_0 : T]

(R (true ⊢ ∃ x . x' = w_0 ∧ w_0 : T ∧ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [Predicate Calculus: x' = w_0 ∧ w_0 : T ⇒ x : T]

(R (true ⊢ ∃ x . x' = w_0 ∧ w_0 : T ∧ x : T ∧ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

*⇒ **[Predicate Calculus: x' = w_0 ∧ w_0 : T ∧ x : T ⇒ ∃ x, x' : T]***

(R (true ⊢ ∃ x, x' : T . x' = w_0 ∧ w_0 : T ∧ x : T ∧ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [Predicate Calculus: x, x' and w_0 do not occur on R]

∃ x, x' . x' = w_0 ∧ w_0 : T ∧ x : T ∧ (R (true ⊢ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

*⇒ **[Predicate Calculus: removing x' = w_0, w_0 : T and x : T weakens the expression]***

∃ x, x' : T. (R (true ⊢ v' = v ∧ tr' = tr ∧ ¬ wait') ; Skip ; A)

= [Skip denotational definition B.3]

∃ x, x' : T. (Skip ; Skip ; A)

= [Theorem C.2.2 twice]

∃ x, x' : T . A

= [Variable Block Denotational Definition B.1]

*(**var** x : T • A)* □

Lemma C.9.

$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2) =$

$c_1 \wedge c_2 \Rightarrow$

$(Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_1\ OP\ P_2$

$\sqsubseteq (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_3\ OP\ P_2))$

provided that

$Lift\ (gA\ (P_1)) = Lift\ (gA\ (P_3))$

Proof:

To reduce verbose, we create the abbreviation $gA(s)$ such that

$gA\,(s) = getAssignments\,(s)$

LHS

$=$

$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)$

= [Definition of Syntactic Transition C.27]

$$\forall w \,.\, c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1\ OP\ P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_3\ OP\ P_2))\,;\ P_3\ OP\ P_2)) \\ \wedge\ Lift\,(gA\,(P_1\ OP\ P_2)) = Lift\,(gA\,(P_3\ OP\ P_2)) \end{array} \right)$$

= [w is universally quantified, so we abstract it]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1\ OP\ P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_3\ OP\ P_2))\,;\ P_3\ OP\ P_2)) \\ \wedge\ Lift\,(gA\,(P_1\ OP\ P_2)) = Lift\,(gA\,(P_3\ OP\ P_2)) \end{array} \right)$$

= [Definition of getAssignments C.1 (4x) for binary Operator OP]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_3) \wedge gA\,(P_2))\,;\ P_3\ OP\ P_2)) \\ \wedge\ Lift\,(gA\,(P_1) \wedge gA\,(P_2)) = Lift\,(gA\,(P_3) \wedge gA\,(P_2)) \end{array} \right)$$

= [Lift And C.2.48]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq\ Lift\,(gA\,(P_3) \wedge gA\,(P_2))\,;\ P_3\ OP\ P_2)) \\ \wedge\ Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2)) = Lift\,(gA\,(P_3)) \wedge Lift\,(gA\,(P_2)) \end{array} \right)$$

= [Assumption]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_3) \wedge gA\,(P_2))\,;\ P_3\ OP\ P_2)) \\ \wedge\ Lift\,(gA\,(P_3)) \wedge Lift\,(gA\,(P_2)) = Lift\,(gA\,(P_3)) \wedge Lift\,(gA\,(P_2)) \end{array} \right)$$

= [Predicate Calculus: (P = P) = true]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_3) \wedge gA\,(P_2))\,;\ P_3\ OP\ P_2)) \\ \wedge\ true \end{array} \right)$$

= [Predicate Calculus]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_3) \wedge gA\,(P_2))\,;\ P_3\ OP\ P_2)) \end{array} \right)$$

= [Lift And C.2.48]

$c_1 \wedge c_2 \Rightarrow$
$(Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_1\ OP\ P_2 \sqsubseteq (Lift\,(gA\,(P_3)) \wedge Lift\,(gA\,(P_2));\ P_3\ OP\ P_2))$

= [Assumption]

$c_1 \wedge c_2 \Rightarrow$
$(Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_1\ OP\ P_2 \sqsubseteq (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_3\ OP\ P_2))$

$=$

RHS

Lemma C.10. $(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_1\ OP\ P_3) =$

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_1\ OP\ P_2 \sqsubseteq (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2));\ P_1\ OP\ P_3)) \end{array} \right)$$

provided that Lift (gA (P_2)) = Lift (gA (P_3))

Proof:

To reduce verbose, we create the abbreviation $gA(s)$ such that
$gA(s) = getAssignments(s)$

LHS

$=$

$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_1\ OP\ P_3)$

= [Definition of Syntactic Transition C.27]

$$\forall\, w\,.\, c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1\ OP\ P_2));\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_1\ OP\ P_3));\ P_1\ OP\ P_3)) \\ \wedge\ Lift\,(gA\,(P_1\ OP\ P_2)) = Lift\,(gA\,(P_1\ OP\ P_3)) \end{array} \right)$$

= [w is universally quantified, so we abstract it]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1\ OP\ P_2));\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_1\ OP\ P_3));\ P_1\ OP\ P_3)) \\ \wedge\ Lift\,(gA\,(P_1\ OP\ P_2)) = Lift\,(gA\,(P_1\ OP\ P_3)) \end{array} \right)$$

= [Definition of getAssignments C.1]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2));\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_1) \wedge gA\,(P_3));\ P_1\ OP\ P_3)) \\ \wedge\ Lift\,(gA\,(P_1) \wedge gA\,(P_2)) = Lift\,(gA\,(P_1) \wedge gA\,(P_3)) \end{array} \right)$$

= [Lift And C.2.48]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2));\ P_1\ OP\ P_2 \sqsubseteq Lift\,(gA\,(P_1) \wedge gA\,(P_3));\ P_1\ OP\ P_3)) \\ \wedge\ Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2)) = Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_3)) \end{array} \right)$$

= [Assumption]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\; P_1\; OP\; P_2 \sqsubseteq Lift\,(gA\,(P_1) \wedge gA\,(P_3))\,;\; P_1\, OP\, P_3)) \\ \wedge\; Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_3)) = Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_3)) \end{array} \right)$$

= [Predicate Calculus: (P = P) = true]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\; P_1\; OP\; P_2 \sqsubseteq Lift\,(gA\,(P_1) \wedge gA\,(P_3))\,;\; P_1\, OP\, P_3)) \\ \wedge\; true \end{array} \right)$$

= [Predicate Calculus: (P $\wedge$ true) = P]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1) \wedge gA\,(P_2))\,;\; P_1\; OP\; P_2 \sqsubseteq Lift\,(gA\,(P_1) \wedge gA\,(P_3))\,;\; P_1\, OP\, P_3)) \end{array} \right)$$

= [Lift And C.2.48]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2))\,;\; P_1\; OP\; P_2 \sqsubseteq (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_3))\,;\; P_1\, OP\, P_3)) \end{array} \right)$$

= [Assumption]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2))\,;\; P_1\; OP\; P_2 \sqsubseteq (Lift\,(gA\,(P_1)) \wedge Lift\,(gA\,(P_2))\,;\; P_1\, OP\, P_3)) \end{array} \right)$$

=

RHS

Lemma C.11. $true \sqsubseteq X$

Proof

$true \sqsubseteq X$

$= [X \Rightarrow true]$

$= [true] = true$

□

Lemma C.12. **if** $(pred_1) \rightarrow A_1 \;[]\; \ldots \;[]\; (pred_n) \rightarrow A_n$ **fi** $\sqsubseteq A1$

provided that

- $pred_1$

***Proof**:*

if $(pred_1) \rightarrow A_1 \;[]\; \ldots \;[]\; (pred_n) \rightarrow A_n$ **fi** $\sqsubseteq A1$

= [If-Guarded Command Denotational Definition B.5]

$$
\begin{array}{l}
R\ (\\
(pred_1 \lor ... \lor pred_n) \\
\land\ (pred_1 \Rightarrow \neg\ {A_1}^{\ f}_{\ f} \land ... \land pred_n \Rightarrow \neg\ {A_n}^{\ f}_{\ f}) \vdash ((pred_1 \land {A_1}^{\ t}_{\ f}) \lor ... \lor (pred_n \land {A_n}^{\ t}_{\ f}) \\
)) \sqsubseteq A1
\end{array}
$$

= [Assumption $pred_1$]

$$
\begin{array}{l}
(R\ (\\
(true \lor ... \lor pred_n) \\
\land\ (true \Rightarrow \neg\ {A_1}^{\ f}_{\ f} \land ... \land pred_n \Rightarrow \neg\ {A_n}^{\ f}_{\ f}) \vdash ((true \land {A_1}^{\ t}_{\ f}) \lor ... \lor (pred_n \land {A_n}^{\ t}_{\ f}) \\
))) \sqsubseteq A1
\end{array}
$$

= [Predicate Calculus]

$$
\begin{array}{l}
(R\ (\\
true \\
\land\ (true \Rightarrow \neg\ {A_1}^{\ f}_{\ f} \land ... \land pred_n \Rightarrow \neg\ {A_n}^{\ f}_{\ f}) \vdash ((true \land {A_1}^{\ t}_{\ f}) \lor ... \lor (pred_n \land {A_n}^{\ t}_{\ f}) \\
))) \sqsubseteq A1
\end{array}
$$

= [Predicate Calculus]

$$
\begin{array}{l}
(R\ ((true \Rightarrow \neg\ {A_1}^{\ f}_{\ f} \land ... \land pred_n \Rightarrow \neg\ {A_n}^{\ f}_{\ f}) \vdash ((true \land {A_1}^{\ t}_{\ f}) \lor ... \lor (pred_n \land {A_n}^{\ t}_{\ f}) \\
))) \sqsubseteq A1
\end{array}
$$

= [Predicate Calculus]

$$
\begin{array}{l}
(R\ ((\neg\ {A_1}^{\ f}_{\ f} \land ... \land pred_n \Rightarrow \neg\ {A_n}^{\ f}_{\ f}) \vdash ({A_1}^{\ t}_{\ f} \lor ... \lor (pred_n \land {A_n}^{\ t}_{\ f}) \\
))) \sqsubseteq A1
\end{array}
$$

= [Disjunction Reactive Designs C.36]

$$
\begin{array}{l}
(R\ (\neg\ {A_1}^{\ f}_{\ f} \vdash {A_1}^{\ t}_{\ f}) \lor R\ (\neg\ (pred_2 \Rightarrow {A_2}^{\ f}_{\ f}) \vdash (pred_2 \land {A_2}^{\ t}_{\ f})) ... \lor R\ (\neg\ (pred_2 \Rightarrow {A_2}^{\ f}_{\ f}) \vdash \\
(pred_2 \land {A_2}^{\ t}_{\ f})) \\
) \sqsubseteq A1
\end{array}
$$

= [Refinement Definition C.14]

$$
\left[\begin{array}{l}
A1 \Rightarrow \\
\left(\begin{array}{l}
R(\neg\ {A_1}^{\ f}_{\ f} \vdash {A_1}^{\ t}_{\ f}) \lor R(\neg\ (pred_2 \Rightarrow {A_2}^{\ f}_{\ f}) \vdash (pred_2 \land {A_2}^{\ t}_{\ f})) \lor ... \\
\lor R(\neg\ (pred_n \Rightarrow {A_n}^{\ f}_{\ f}) \vdash (pred_n \land {A_n}^{\ t}_{\ f}))
\end{array}\right)
\end{array}\right]
$$

= [CSP process as a Self-Reactive Design 12, having, from theorem 13, that it can be applied to Circus Actions]

$$
\left[\begin{array}{l}
A1 \Rightarrow \\
\left(\begin{array}{l}
A1 \lor R(\neg\ (pred_2 \Rightarrow {A_2}^{\ f}_{\ f}) \vdash (pred_2 \land {A_2}^{\ t}_{\ f})) \lor ... \\
\lor R(\neg\ (pred_n \Rightarrow {A_n}^{\ f}_{\ f}) \vdash (pred_n \land {A_n}^{\ t}_{\ f}))
\end{array}\right)
\end{array}\right]
$$

= [Predicate Calculus]

$$\left[\begin{array}{l} \neg A1 \vee \\ \left(\begin{array}{l} A1 \vee R(\neg\,(pred_2 \Rightarrow A_2{}^{f}_{f}) \vdash (pred_2 \wedge A_2{}^{t}_{f})) \vee \ldots \\ \vee R(\neg\,(pred_n \Rightarrow A_n{}^{f}_{f}) \vdash (pred_n \wedge A_n{}^{t}_{f})) \end{array}\right) \end{array}\right]$$

= [Predicate Calculus]

$$\left[\begin{array}{l} \neg A1 \vee A1 \vee \\ \left(\begin{array}{l} R(\neg\,(pred_2 \Rightarrow A_2{}^{f}_{f}) \vdash (pred_2 \wedge A_2{}^{t}_{f})) \vee \ldots \\ \vee R(\neg\,(pred_n \Rightarrow A_n{}^{f}_{f}) \vdash (pred_n \wedge A_n{}^{t}_{f})) \end{array}\right) \end{array}\right]$$

= [Predicate Calculus]

$$\left[\begin{array}{l} true \vee \\ \left(\begin{array}{l} R(\neg\,(pred_2 \Rightarrow A_2{}^{f}_{f}) \vdash (pred_2 \wedge A_2{}^{t}_{f})) \vee \ldots \\ \vee R(\neg\,(pred_n \Rightarrow A_n{}^{f}_{f}) \vdash (pred_n \wedge A_n{}^{t}_{f})) \end{array}\right) \end{array}\right]$$

= [Predicate Calculus]

$[true]$

= [Predicate Calculus]

true

Lemma C.13. ***Refinement Conjunctive Monotonic***

$(P \wedge F \sqsubseteq Q \wedge F)$

provided that $(P \sqsubseteq Q)$

Proof*:*

For F = true:

$(P \wedge true \sqsubseteq Q \wedge true)$

= [Predicate Calculus]

$(P \sqsubseteq Q)$

= [Assumption]

true

For F = false:

$(P \wedge false \sqsubseteq Q \wedge false)$

= [Predicate Calculus]

$(false \sqsubseteq false)$

= [Refinement Definition C.14]

$[(false \Rightarrow false)]$

= [Predicate Calculus]

true

Lemma C.14. ***Design Trace R2*** *(true ⊢ s ∧ tr' = tr0 ∧ ¬ wait') is R2*
Proof*:*
(true ⊢ s ∧ tr' = tr0 ∧ ¬ wait')
= [Sequence Property]
(true ⊢ s ∧ tr' - tr0 = ⟨⟩ ∧ ¬ wait')
= [Definition of R2 C.20]
R2 (true ⊢ s ∧ tr' = tr0 ∧ ¬ wait')

Lemma C.15. ****Existential Quantifier Shifted Inside R:***
∃ x . R (P) = R (∃ x . P),

provided that x is neither tr nor wait

This theorem lies on (Oliveira 2006*b*).

Lemma C.16. ***Existential Quantifier Distributed Throughout Design:***
∃ x . (true ⊢ Post) = (true ⊢ (∃ x . Post)),

provided that x is neither ok or ok'.

Proof:

∃ x . (true ⊢ Post)
[Definition of Design C.23]
∃ x . ((true ∧ ok) ⇒ (Post ∧ ok'))
[Predicate Calculus]
∃ x . (ok ⇒ (Post ∧ ok'))
[Predicate Calculus]
∃ x . (¬ ok ∨ (Post ∧ ok'))
[Predicate Calculus]
(∃ x . ¬ ok) ∨ (∃ x . Post ∧ ok')
[Predicate Calculus]

$(\exists\, x \,.\, \neg\, (true \wedge ok)) \vee (\exists\, x \,.\, Post \wedge ok')$
[Predicate Calculus]
$(\neg\ \forall\, x \,.\, (true \wedge ok)) \vee (\exists\, x \,.\, Post \wedge ok')$
[Predicate Calculus]
$(\forall\, x \,.\, (true \wedge ok)) \Rightarrow (\exists\, x \,.\, Post \wedge ok')$
[Predicate Calculus]
$(\forall\, x \,.\, (true \wedge ok)) \Rightarrow (\exists\, x \,.\, Post \wedge ok')$
[Predicate Calculus]
$((true \wedge \forall\, x \,.\, ok)) \Rightarrow (\exists\, x \,.\, Post \wedge ok')$
[Assumption]
$((true \wedge ok)) \Rightarrow (\exists\, x \,.\, Post \wedge ok')$
[Assumption]
$((true \wedge ok)) \Rightarrow ((\exists\, x \,.\, Post) \wedge ok')$
[Definition of Design C.23]
$true \vdash (\exists\, x \,.\, Post)$

Lemma C.17. ***Labelled Transition Implication***
$(c_1 \mid s_1 \models A_1) \xrightarrow{d^* w_1} (c_3 \mid s_3 \models A_3)$
$\Rightarrow$
$(Lift\ (s_1);\ A_1) \sqsubseteq (Lift\ (s_3);\ d^* w_1 \rightarrow A_3)$

Proof:

$(c_1 \mid s_1 \models A_1) \xrightarrow{d^* w_1} (c_3 \mid s_3 \models A_3)$
= [Definition of Labelled Transition C.26]
$\forall\, w.\ c_1 \wedge c_3 \Rightarrow (Lift\ (s_1);\ A_1) \sqsubseteq (Lift\ (s_3);\ d^* w_1 \rightarrow A_3) \,\Box\, (Lift\ (s_1);\ A_1)$
= [w is universally quantified, so we abstract it]
$c_1 \wedge c_3 \Rightarrow (Lift\ (s_1);\ A_1) \sqsubseteq (Lift\ (s_3);\ d^* w_1 \rightarrow A_3) \,\Box\, (Lift\ (s_1);\ A_1)$
$\Rightarrow$ ***[External Choice Refinement Implication 15]***
$c_1 \wedge c_3 \Rightarrow (Lift\ (s_1);\ A_1) \sqsubseteq (Lift\ (s_3);\ d^* w_1 \rightarrow A_3)$

C.2.3 Auxiliary Theorems

Theorem 1. ***Refinement Equal Sides:***
$A \sqsubseteq A$

Proof

$A \sqsubseteq A$

= [Definition of Refinement]

$[A \Rightarrow A]$

= [Predicate Calculus]

$[\neg A \vee A]$

= [Predicate Calculus]

[true]

= [Predicate Calculus]

true

□

Theorem 2.

Silent Transition between equivalent nodes:

$(c \mid s \models A) \xrightarrow{\tau} (c \mid s \models A)$

Proof

$(c \mid s \models A) \xrightarrow{\tau} (c \mid s \models A)$

= [Definition of Silent Transition, C.25]

$\forall w . c \wedge c \Rightarrow ((\text{Lift } (s) ; A \sqsubseteq \text{Lift } (s) ; A)$

= [Refinement Equal Sides 1]

$\forall w. c \wedge c \Rightarrow \text{true}$

= [(P $\Rightarrow$ true) = true]

$\forall w. \text{true}$

=

true

□

Theorem 3.

Sigma Equal:

$(c \models P) \rightsquigarrow (c \models P)$

Proof $(c \models P) \rightsquigarrow (c \models P)$

= [Definition of Sigma Transition]

$$\forall w.c \wedge c \Rightarrow \left(\begin{array}{l} (Lift(getAssignments(P)); P \sqsubseteq Lift(getAssignments(P)); P) \\ \wedge Lift(getAssignments(P)) = Lift(getAssignments(P)) \end{array} \right)$$

= [Refinement Equal Sides (2x) 1]

$\forall$ w . c $\wedge$ c $\Rightarrow$ (true $\wedge$ true)
= [Predicate Calculus]
$\forall$ w. c $\wedge$ c $\Rightarrow$ true
= [Predicate Calculus]
true □

Theorem 4.
***Sigma Equal Basic Process**:*
$(c \models P) \rightsquigarrow (c \mid s \models B)$

provided that $P = B$

Proof
(c $\models$ P) $\rightsquigarrow$ (c | s $\models$ B)
= [Definition of Syntactic Transition C.27]
$$\forall w.c \wedge c \Rightarrow \left(\begin{array}{l} (Lift(getAssignments(P));\ P \sqsubseteq Lift(s);\ B) \\ \wedge\ Lift(getAssignments(P)) = Lift(s) \end{array} \right)$$
= [Assumption P = B]
$$\forall w.c \wedge c \Rightarrow \left(\begin{array}{l} (Lift(getAssignments(B));\ B \sqsubseteq Lift(s);\ B) \\ \wedge\ Lift(getAssignments(B)) = Lift(s) \end{array} \right)$$
= [Definition of getAssignments C.1]
$$\forall w.c \wedge c \Rightarrow \left(\begin{array}{l} (Lift(s);\ B \sqsubseteq Lift(s);\ B) \\ \wedge\ Lift(s) = Lift(s) \end{array} \right)$$
= [Predicate Calculus]
$$\forall w.c \wedge c \Rightarrow \left(\begin{array}{l} (Lift(s);\ B \sqsubseteq Lift(s);\ B) \\ \wedge\ true \end{array} \right)$$
= [Refinement Equal Sides 1]
$$\forall w.c \wedge c \Rightarrow \left(\begin{array}{l} true \\ \wedge\ true \end{array} \right)$$
= [Predicate Calculus]
true □

Theorem 5.

***S Sequence False**:*
s ; $false = false$

This theorem is quoted on (Hoare & Jifeng 1998).

Theorem 6.

***Monotonicity of Action Refinement**:*
A_1 ; $A_2 \sqsubseteq B_1$; B_2 *provided* $A_1 \sqsubseteq B_1$ *and* $A_2 \sqsubseteq B_2$

Proof The proof lies on the document (Oliveira 2006*b*).
□

Theorem 7.
***Lift Shift**:*
$l \rightarrow Lift\ (s)$; $A = Lift\ (s)$; $l \rightarrow A$

provided that

- $ok = ok'$

Where FV (l) are the free variables of l, and LHS (s) returns the left variables of the sequence s of assignments. We use Theorem 11 to assume that all Lifted assignments (e.g. Lift (s)) are Circus actions.

Proof
l $\rightarrow$ Lift (s) ; A
= [Prefixing Sequence C.2.5, having, from theorem 11, that Lift (s) is a *Circus* action]
l $\rightarrow$ Skip ; Lift (s) ; A
= [Definition of Lift (C.24)]
l $\rightarrow$ Skip ; (R1 $\circ$ R3 (true $\vdash$ s $\wedge$ tr' = tr $\wedge$ $\neg$ wait')) ; A
= [Sequence Denotational Definition C.33 and Prefixing Denotational Definition B.2 (where v0 is a set that contains all observational variables in an intermediate state (ok0, wait0, ref0, tr0 and so on))]

$$\exists v0. \left(\begin{array}{l} R(true \vdash doC(l,C) \wedge vars0 = vars) \\ \wedge\ (R1 \circ R3(true \vdash s(\alpha 0, \alpha') \wedge tr' = tr0 \wedge \neg\, wait')) \end{array} \right) ; A$$

= [Definition of doC C.32]

$$\exists v0. \left(\begin{array}{l} R\left(true \vdash \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array} \right) \wedge vars0 = vars \right) \\ \wedge\ (R1 \circ R3(true \vdash s(\alpha 0, \alpha') \wedge tr' = tr0 \wedge \neg\, wait')) \end{array} \right) ; A$$

= [Design Trace R2 C.14]

$$\exists v0.\left(\begin{array}{l} R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \end{array}\right) \\ \wedge \left(R(true \vdash s(\alpha 0, \alpha') \wedge tr' = tr0 \wedge \neg\, wait')\right) \end{array}\right) ; A$$

= [State Assignment C.16]

$$\exists v0.\left(\begin{array}{l} R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \end{array}\right) \\ \wedge \left(R\left(true \vdash \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge wait' = wait0 \\ \wedge\, ok' = ok0 \wedge tr' = tr0 \wedge ref' = ref0 \\ \wedge\, nalocalvars' = nalocalvars0 \\ \wedge\, tr' = tr0 \wedge \neg\, wait' \end{array}\right)\right)\right) \end{array}\right) ; A$$

= [Closure Conjunctive R C.2.42]

$$\exists v0.\left(\begin{array}{l} R\left(\begin{array}{l} \left(true \vdash \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars\right) \\ \wedge \left(true \vdash \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge wait' = wait0 \\ \wedge\, ok' = ok0 \wedge tr' = tr0 \\ \wedge\, ref' = ref0 \\ \wedge\, nalocalvars' = nalocalvars0 \\ \wedge\, tr' = tr0 \wedge \neg\, wait' \end{array}\right)\right) \end{array}\right) \end{array}\right) ; A$$

= [Designs And True C.2.46]

$$\exists v0.\left(\begin{array}{l} R\left(true \vdash \left(\begin{array}{l} \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge f(\alpha 0, \alpha') \wedge wait' = wait0 \\ \wedge\, ok' = ok0 \wedge tr' = tr0 \\ \wedge\, ref' = ref0 \\ \wedge\, nalocalvars' = nalocalvars0 \\ \wedge\, tr' = tr0 \wedge \neg\, wait' \end{array}\right)\right) \end{array}\right) ; A$$

= [Predicate Calculus: rearranging the order of the and operands and parenthesis]

$$\exists v0.\ \left(\ R\ \left(\ true \vdash \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge\ (wait' = wait0) \wedge \neg\, wait' \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \\ \wedge\ (tr' = tr0) \\ \wedge\ ok' = ok0 \end{array}\right)\ \right)\ \right)\ ;\ A$$

= [Existential Quantifier Shifted Inside R C.15]

$$R\ \left(\ \exists v0.\ \left(\ true \vdash \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge\ (wait' = wait0) \wedge \neg\, wait0 \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \end{array}\right)\ \right)\ \right)\ ;\ A$$

= [Existential Quantifier Distributed Throughout Design C.16]

$$R\ \left(\ true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge\ (wait' = wait0) \wedge \neg\, wait0 \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \end{array}\right)\ \right)\ ;\ A$$

= [Predicate Calculus: One-point-rule]

$$R\ \left(\ true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge\ (wait' = wait0) \wedge \neg\, wait0 \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \end{array}\right)\ \right)\ ;\ A$$

= [Predicate Calculus: replacing $\neg\ wait0$]

$$R\left(\begin{array}{l} true \vdash \exists v0. \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge \\ \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge\ (wait' = wait0) \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: One-point-rule (inserting wait = wait0)]

$$R\left(\begin{array}{l} true \vdash \exists v0. \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge wait = wait0 \wedge \\ \left(\begin{array}{l} tr0 = tr \wedge (l,C) \notin ref0 \\ \lhd wait0 \rhd \\ tr0 = tr \frown \langle (l,C) \rangle \end{array}\right) \wedge vars0 = vars \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: One-point-rule (replacing wait0 by wait' on the conditional)]

$$R\left(\begin{array}{l} true \vdash \exists v0. \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge wait = wait0 \wedge \\ ((tr0 = tr \wedge (l,C) \notin ref0 \lhd wait' \rhd tr0 = tr \frown \langle (l,C) \rangle) \wedge vars0 = vars) \\ \wedge\ (ref' = ref0) \wedge (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: One-point-rule (inserting ref = ref0)]

$$R\left(\begin{array}{l} true \vdash \exists v0. \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge wait = wait0 \wedge ref = ref0 \wedge \\ ((tr0 = tr \wedge (l,C) \notin ref0 \lhd wait' \rhd tr0 = tr \frown \langle (l,C) \rangle) \wedge vars0 = vars) \\ \wedge\ (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: One-point-rule (replacing $\notin ref0$ by $\notin ref'$)]

$$R\left(\begin{array}{l} true \vdash \exists v0. \left(\begin{array}{l} f(\alpha 0, \alpha') \wedge wait = wait0 \wedge ref = ref0 \wedge \\ ((tr0 = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr0 = tr \frown \langle (l,C) \rangle) \wedge vars0 = vars) \\ \wedge\ (nalocalvars' = nalocalvars0) \\ \wedge\ (tr' = tr0) \wedge (tr' = tr0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: And Idempotence (tr' = tr0) and commutative]

$$R\left(\begin{array}{l} true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge wait = wait0 \wedge ref = ref0 \wedge \\ ((tr0 = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr0 = tr \frown \langle (l,C) \rangle) \wedge vars0 = vars) \\ \wedge\ (tr' = tr0) \\ \wedge\ (nalocalvars' = nalocalvars0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right); A$$

= [Predicate Calculus: One-point-rule (inserting tr = tr0)]

$$R\left(\begin{array}{l} true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ ((tr0 = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr0 = tr \frown \langle (l,C) \rangle) \wedge vars0 = vars) \\ \wedge\ (tr' = tr0) \\ (nalocalvars' = nalocalvars0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right); A$$

= [Predicate Calculus: One-point-rule (removing tr' = tr0)]

$$R\left(\begin{array}{l} true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ ((tr0 = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr0 = tr \frown \langle (l,C) \rangle) \wedge vars0 = vars) \\ \wedge\ (nalocalvars' = nalocalvars0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right); A$$

= [By the def. of Prefixing Denotational Definition B.2, vars0 = vars includes all local variables (some of which are the assignments of nalocalvars), so nalocalvars $\subseteq$ vars. Thus, vars0 = vars can be rewritten as f (α, $\alpha 0$) $\wedge$ nalocalvars0 = nalocalvars, such that f (α, $\alpha 0$) does not change its values]

$$R\left(\begin{array}{l} true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ f(\alpha,\alpha 0) \wedge nalocalvars0 = nalocalvars \end{array}\right) \\ \wedge\ (nalocalvars' = nalocalvars0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right); A$$

= [Predicate Calculus: $f(\alpha 0,\alpha')$ and $f(\alpha,\alpha 0)$ side by side]

$$R\left(\begin{array}{l} true \vdash \exists v0.\ \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge f(\alpha,\alpha 0) \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ nalocalvars0 = nalocalvars \end{array}\right) \\ \wedge\ (nalocalvars' = nalocalvars0) \\ \wedge\ ok' = ok0 \wedge \neg\, wait0 \end{array}\right) \end{array}\right); A$$

= [Predicate Calculus: switching the positions of *nalocalvars0 = nalocalvars* and *nalocalvars' = nalocalvars0*]

$$R\left(\begin{array}{l} true \vdash \exists v0.\left(\begin{array}{l} f(\alpha 0,\alpha') \wedge f(\alpha,\alpha 0) \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \\ \wedge\ nalocalvars0 = nalocalvars \\ \wedge\ ok' = ok0 \wedge \neg\ wait0 \end{array}\right) \end{array}\right) ; A$$

= [Assumption: ok = ok']

$$R\left(\begin{array}{l} true \vdash \exists v0.\left(\begin{array}{l} f(\alpha 0,\alpha') \wedge f(\alpha,\alpha 0) \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \\ \wedge\ nalocalvars0 = nalocalvars \wedge ok = ok0 \wedge \neg\ wait0 \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: placing *nalocalvars0 = nalocalvars* $\wedge$ *ok = ok0* $\wedge$ $\neg$ *wait0* above]

$$R\left(\begin{array}{l} true \vdash \exists v0.\left(\begin{array}{l} nalocalvars0 = nalocalvars \wedge ok = ok0 \wedge \neg\ wait0 \wedge \\ f(\alpha 0,\alpha') \wedge f(\alpha,\alpha 0) \wedge wait = wait0 \wedge ref = ref0 \wedge tr = tr0 \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right) ; A$$

= [Predicate Calculus: rearranging some factors]

$$R\left(\begin{array}{l} true \vdash \exists v0.\left(\begin{array}{l} f(\alpha,\alpha 0) \\ \wedge\ ok = ok0 \wedge wait = wait0 \wedge ref = ref0 \\ \wedge\ nalocalvars0 = nalocalvars \wedge tr0 = tr \wedge \neg\ wait0 \\ \wedge f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right) ; A$$

= [Existential Quantifier Distributed Throughout Design C.16]

$$R\left(\begin{array}{l} \exists v0.true \vdash \left(\begin{array}{l} f(\alpha,\alpha 0) \\ \wedge\ ok = ok0 \wedge wait = wait0 \wedge ref = ref0 \\ \wedge\ nalocalvars0 = nalocalvars \wedge tr0 = tr \wedge \neg\ wait0 \\ \wedge f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right) ; A$$

= [Existential Quantifier Shifted Inside R (reverse) C.15]

$$\exists v0.R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} f(\alpha,\alpha 0) \\ \wedge\ ok = ok0 \wedge wait = wait0 \wedge ref = ref0 \\ \wedge\ nalocalvars0 = nalocalvars \wedge tr0 = tr \wedge \neg\ wait0 \\ \wedge f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right) ; A$$

= [Designs And True C.2.46]

$$\exists v0.R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} f(\alpha,\alpha 0) \\ \wedge\ ok = ok0 \wedge wait = wait0 \wedge ref = ref0 \\ \wedge\ nalocalvars0 = nalocalvars \wedge tr = tr0 \wedge \neg\ wait0 \end{array}\right) \\ \wedge\ true \vdash \left(\begin{array}{l} \wedge f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right); A$$

= [State Assignment C.16]

$$\exists v0.R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} s(\alpha,\alpha 0) \wedge tr = tr0 \wedge \neg\ wait0 \end{array}\right) \\ \wedge\ true \vdash \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right); A$$

= [Closure Conjunctive R C.2.42]

$$\exists v0.\left(\begin{array}{l} R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} s(\alpha,\alpha 0) \wedge tr = tr0 \wedge \neg\ wait0 \end{array}\right) \end{array}\right) \\ \wedge\ R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} f(\alpha 0,\alpha') \wedge \\ \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right); A$$

= [Predicate Calculus: repositioning nalocalvars' = nalocalvars0]

$$\exists v0.\left(\begin{array}{l} R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} s(\alpha,\alpha 0) \wedge tr = tr0 \wedge \neg\ wait0 \end{array}\right) \end{array}\right) \\ \wedge\ R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge f(\alpha 0,\alpha') \wedge (nalocalvars' = nalocalvars0) \end{array}\right) \end{array}\right) \end{array}\right); A$$

= [On the denotational definition of Prefixing B.2, f (α0, α′) ∧ nalocalvars' = nalocalvars0 is vars' = vars0]

$$\exists v0.\left(\begin{array}{l} R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} s(\alpha,\alpha 0) \wedge tr = tr0 \wedge \neg\ wait0 \end{array}\right) \end{array}\right) \\ \wedge\ R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ vars' = vars0 \end{array}\right) \end{array}\right) \end{array}\right); A$$

= [Definition of Lift C.24]

$$\exists v0.\left(\begin{array}{l} Lift(s)(\alpha,\alpha 0) \\ \wedge\ R\left(\begin{array}{l} true \vdash \left(\begin{array}{l} (tr' = tr \wedge (l,C) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (l,C) \rangle) \\ \wedge\ vars' = vars0 \end{array}\right) \end{array}\right) \end{array}\right); A$$

= [Definition of doC C.32]

$$\exists v0.\left(\begin{array}{l} (Lift(s))(\alpha,\alpha 0) \\ \wedge\ (l \rightarrow Skip)(\alpha 0,\alpha') \end{array}\right); A$$

= [Sequence Denotational Definition C.33]

Lift (s) ; l → Skip

□

Theorem 8. ***Parallel Prefixed Loc****:*

(loc s1 • A1) $[\![$ *x1 | CS | x2* $]\!]$ *(loc s2 • A2)*
$\sqsubseteq$
$l \rightarrow$ *((loc s3 • A3)* $[\![$ *x1 | CS | x2* $]\!]$ *(loc s2 • A2))*

provided that

Lift (s1) ; A1 $\sqsubseteq$ *Lift (s3) ;* $l \rightarrow$ *A3* $\Box$ *Lift (s1) ; A1* ***[Assumption 1]***
(assumption 1 is the right side of the denotational definition of labelled transition
$(c1 \mid s1 \models A1) \xrightarrow{l} (c3 \mid s3 \models A3)$*)*

initials (Lift (s2) ; A2) $\subseteq$ *CS* ***[Assumption 2]***
(assumption 2 indicates that we require that the initial events being offered by Lift (s2) ; A2 lie on channel set CS)

CS $\cap$ *usedC (Lift (s3); A3)* $= \emptyset$ ***[Assumption 3]***
(assumption 3 demands that no channel used on Lift (s3); A3 can lie on channel set CS)

wrtV (Lift (s3); A3) $\cap$ *usedV (Lift (s2) ; A2)* $= \emptyset$ ***[Assumption 4]***
(assumption 4 demands that no variables that are used by action Lift (s2) ; A2 can be written)

Lift (s2) ; A2 is divergence free ***[Assumption 5]***
(assumption 5 is self-explanatory)

ok = ok' ***[Assumption 6]***
(assumption 6 demands that either (1) if the program has started than it has finished, or (2) if the program did not start then it did not finish)

Proof
Before starting to prove, we will derive a new assumption from the provided assumption:
[Assumption 1] Lift (s1); A1 $\sqsubseteq$ Lift (s3); l $\rightarrow$ A3 $\Box$ Lift (s1); A1
$\Rightarrow$ **[External Choice Refinement Implication 15]**
[Assumption ppl] Lift (s1) ; A1 $\sqsubseteq$ Lift (s3) ; l $\rightarrow$ A3

We also assume Theorem 11 on the proof to assure that Lift (s1), Lift (s2) and Lift (s3) are *Circus* actions. We will add an explanation of each tactic and sub-goal, envisaging explaining this proof. Now we will start proving:
((loc s1 • A1) $[\![$ x1 | CS | x2 $]\!]$ (loc s2 • A2))
= [Definition of loc C.29]
(We will apply the definitions of loc in order to convert them to Lift expressions. Lift expressions allow the application of the lemmas and theorems used throughtout the proof)

((Lift (s1) ; A1) ⟦ x1 ∣ CS ∣ x2 ⟧ (Lift (s2) ; A2))
⊑ **[Assumption ppl with Parallelism Refinement Monotonic C.2.21]**
(application of assumption ppl and Monotonicity of Refinement for Parallelism allow refining the expression to an expression that has l → A3, linking the left expression to the right expression)
((Lift (s3) ; l → A3) ⟦ x1 ∣ CS ∣ x2 ⟧ (Lift (s2) ; A2))
= [Lift Shift (having Assumption 6) 7]
(Lift Shift has an important role on this proof, as it allows swapping the positions from Lift (s3) and l on the prefixing (→) operator. This is important because the right expression of the refinement we want to prove has l before expression loc s3 • A3)
((l → Lift (s3) ; A3) ⟦ x1 ∣ CS ∣ x2 ⟧ (Lift (s2) ; A2))
= [Prefixing Sequence theorem: C.2.5]
(now we transform l → Lift (s3) into l → Skip ; Lift (s3), in order to allow the application of Parallel Composition Sequence Step)
((l → Skip ; Lift (s3) ; A3) ⟦ x1 ∣ CS ∣ x2 ⟧ (Lift (s2) ; A2))
= [Parallel Composition Sequence Step (having assumptions 2 to 5) C.2.3]
(the role of this step is to take l outside the left branch, making it a prefix not only from the left branch of the parallel composition, but also from the whole parallel composition)
l → ((Lift (s3) ; A3) ⟦ x1 ∣ CS ∣ x2 ⟧ (Lift (s2) ; A2))
= [Definition of loc C.29]
(now we apply again the definition of loc in order to reach the right expression of the refinement)
l → ((loc s3 • A3) ⟦ x1 ∣ CS ∣ x2 ⟧ (loc s2 • A2))
□

Theorem 9. *Parallel Independent Refinement:*
(loc s1 • A1) ⟦ *x1 ∣ CS ∣ x2* ⟧ *(loc s2 • A2)* ⊑ *(loc s3 • A3)* ⟦ *x1 ∣ CS ∣ x2* ⟧ *(loc s2 • A2)*

provided that

- *Lift (s1) ; A1* ⊑ *Lift (s3) ; A3*

In order to prove, we use Theorem 11 to assume that all Lift (s_i) are *Circus* actions.

Proof

$$\left(\begin{array}{l}(locs1 \bullet A1) \,[\![x1 \mid CS \mid x2]\!]\, (locs2 \bullet A2) \\ \sqsubseteq (locs3 \bullet A3) \,[\![x1 \mid CS \mid x2]\!]\, (locs2 \bullet A2)\end{array}\right)$$

= [Definition of loc C.29]

$$\left(\begin{array}{l}(Lift(s1);\, A1) \,[\![x1 \mid CS \mid x2]\!]\, (Lift(s2);\, A2) \\ \sqsubseteq (Lift(s3);\, A3) \,[\![x1 \mid CS \mid x2]\!]\, (Lift(s2);\, A2)\end{array}\right)$$

= [Assumption, Ref. Eq. Sides 1 and Refinement Monotonic 6]
true

□

Theorem 10. *Lifted Assignment is a Circus Assignment Command:*
Lift (x := c) = (x := c)

Proof
Lift (x := c)
= [Definition of Lift C.24]
R1 ∘ R3 (true ⊢ x := c ∧ tr' = tr ∧ ¬ wait')
= [Replacing tr' = tr by tr' - tr = ⟨ ⟩]
R1 ∘ R3 (true ⊢ x := c ∧ tr' - tr = ⟨ ⟩ ∧ ¬ wait')
= [The expression is also R2. If it is R1 and R3, it is R]
R (true ⊢ x := c ∧ tr' - tr = ⟨ ⟩ ∧ ¬ wait')
= [State Assignment C.16]
R (true ⊢ x' = c ∧ u' = u ∧ tr' - tr = ⟨ ⟩ ∧ ¬ wait')
= [Assignment Denotational Definition B.4]
x := c □

Theorem 11. *Lifted Sequence of Assignments is a Circus Assignment Command:*
Lift (s) = s

provided that

- s is a sequence of state assignments (C.16) of the type ($x_1 := c_1$; ...; $x_n := c_n$);

Proof
Lift (s)
= [Assumption]
Lift ($x_1 := c_1$; ...; $x_n := c_n$)
= [Lift Composition C.2.27]
Lift ($x_1 := c_1$); ...; Lift ($x_n := c_n$)
= [Lifted Assignment is a Circus Assignment Command 10 (n times)]
$x_1 := c_1$; ...; $x_n := c_n$
= [Assumption]
s
□

Theorem 12. **CSP process as a Self-Reactive Design:*
$A = R\,((\neg\ A_f^f) \vdash (A_f^t))$*, provided that A is a CSP process (CSP1, CSP2 and CSP3-healthy)*

This theorem lies on (Oliveira 2006*a*).

Theorem 13. ****Circus Action CSP1-CSP2-CSP3-Healthy:***
Every Circus action is CSP1 C.38, CSP2 C.39 and CSP3 C.40 Healthy.

This theorem lies on (Oliveira 2006*a*).

Theorem 14.

****Monotonicity of Refinement on External Choice**:*
$A_1 \Box A_2 \sqsubseteq B_1 \Box B_2$ *provided* $A_1 \sqsubseteq B_1$ *and* $A_2 \sqsubseteq B_2$

Proof The proof lies on the document (Oliveira 2006*b*).
□

Theorem 15.

External Choice Refinement Implication
((Lift (s1) ; A1) $\sqsubseteq$ *(Lift (s2) ;* $l \rightarrow$ *A2)* $\Box$ *(Lift (s1) ; A1))*
$\Rightarrow$
((Lift (s1) ; A1) $\sqsubseteq$ *(Lift (s2) ;* $l \rightarrow$ *A2))*

provided that A1 = $l \rightarrow$ *A2*

All lemmas, theorems, rules or laws that use External Choice Refinement Implication automatically provide that A1 = l $\rightarrow$ A2.

Proof
((Lift (s1) ; A1) $\sqsubseteq$ (Lift (s2) ; l $\rightarrow$ A2) $\Box$ (Lift (s1) ; A1))
[Assumption]
((Lift (s1) ; A1) $\sqsubseteq$ (Lift (s2) ; A1) $\Box$ (Lift (s1) ; A1))
$\Rightarrow$ **[Refinement Implication For Subst.Equiv. C.4 (having Lift Subst. Equiv. C.5 both for Lift (s1) and Lift (s2)) + Monotonicity of Refinement for Ext.Choice 14)]**
((Lift (s1) ; A1) $\sqsubseteq$ (Lift (s2) ; A1)

= [Assumption]
((Lift (s1) ; A1) $\sqsubseteq$ (Lift (s2) ; l $\rightarrow$ A2)
□

Theorem 16. ***Hidden Event Sequenced by Skip**:*
((l $\rightarrow$ Skip) $\setminus$ {l}) = Skip

Proof
The proof lies on (Oliveira 2006*a*). □

C.2.4 Auxiliary Laws

On this appendix, we will list all denotational laws that were used throughtout this document (all refinement laws that were used from the Denotational Semantics of *Circus* (Oliveira 2006*a*) (refinement laws), laws concerning the denotational theory of the CML Operational Semantics we have lifted, based on (Cavalcanti & Woodcock 2013), and laws that we created and proved correct).

Refinement Laws

Law C.2.1. *(*) Skip = V : [g, true]*

The above law is shown on (Oliveira 2006*a*).

Law C.2.2. *(*) Skip ; A = A*

Proof The proof lies on the document (Oliveira 2006*a*).
□

Law C.2.3. ***Parallel Composition Sequence Step**:*
(A1 ; A2) [| ns1 {| cs |} ns2 |] A3 = A1 ; (A2 [| ns1 {| cs |} ns2 |] A3)

provided that

- *initials (A3)* $\subseteq$ *cs*
- *cs* $\cap$ *usedC (A1)* $= \emptyset$
- *wrtV (A1)* $\cap$ *usedV (A3)* $= \emptyset$
- *A3 is divergence-free*

Function usedV (Act) is defined at (Oliveira 2006*a*) and returns the set of variables that are used by action Act. Function usedC is defined on C.12. Function wrtV (Act) is defined at (Oliveira 2006*a*) and returns the set of variables that are written by action Act.
Proof The proof lies on the document (Oliveira 2006*a*).
□

Law C.2.4. ***True Guard**: $true \;\&\; A = A$*

Proof The proof lies on the document (Oliveira 2006*a*)
□

Law C.2.5. ***Prefixing Sequence**: $a \rightarrow B = a \rightarrow Skip \;;\; B$*

The proof lies on the document (Oliveira 2006*a*)

Law C.2.6.

***Compound Actions Commutative (Except for Sequence)**:*
A1 BOP A2 = A2 BOP A1
provided that $BOP \in \{ \; [\![\; \{\!| \; cs \; |\!\} \;]\!], \; |||, \; \Box, \; \sqcap, \; \}$

For Parallel Compositions with name-sets, the name-sets are also commuted:
$A1 \; [\![\; ns1 \; \{\!| \; cs \; |\!\} \; ns2 \;]\!] \; A2 = A2 \; [\![\; ns2 \; \{\!| \; cs \; |\!\} \; ns1 \;]\!] \; A1$

Proof There is a version of this law for each compound operator that is commutative, all of which are referenced on document (Oliveira 2006*a*)..

□

Law C.2.7.

***Communication Parallelism Distribution**:*
$c \rightarrow (A1 \; [\![\; \{\!| \; CS \; |\!\} \;]\!] \; A2) = ((c \rightarrow A1) \; [\![\; \{\!| \; CS \; |\!\} \;]\!] \; (c \rightarrow A2))$
provided that $\{ \; c \; \} \in CS$

Proof The law is referenced on document (Oliveira 2006*a*).

□

Law C.2.8. ***Hiding Identity**:*
$A \setminus CS = A$
provided $CS \cap UsedC\ (A) = \emptyset$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.9. ***Hiding Monotonic**:*
$P_1 \setminus S \sqsubseteq P_2 \setminus S$
provided $P_1 \sqsubseteq P_2$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.10. ***Hiding External Choice Distributive**:*
$(A_1 \Box A_2) \setminus cs = (A_1 \setminus cs) \Box (A_2 \setminus cs)$
provided $(initials\ (A1) \cup initials\ (A2)) \cap cs = \emptyset$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.11. ***External Choice Sequence Distributive**:*
$((A_1 \ ; B) \Box (A_2 \ ; B)) = ((A_1 \Box A_2) \ ; B))$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.12. ***Hiding Sequence Distributive**:*
$(A_1 \ ; A_2) \setminus cs = (A_1 \setminus cs) \ ; (A_2 \setminus cs)$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.13. ***Variable Block Extension**:*
$A_1 \ ; (\mathbf{var}\ x : T \bullet A_2) \ ; A_3 = (\mathbf{var}\ x : T \bullet A_1 \ ; A_2 \ ; A_3)$
provided $x \notin FV\ (A_1) \cup FV\ (A_3)$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.14. ***Variable Block Extension (A)**:*
$A_1 \ ; (\mathbf{var}\ x : T \bullet A_2) = (\mathbf{var}\ x : T \bullet A_1 \ ; A_2)$
provided $x \notin FV\ (A_1)$

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.15. ***Circus Basic Process to Circus action**:*
begin state *[decl | pred] PPars • A* **end** $\widehat{=}$ **var** *decl • A*

Proof The law is referenced on document (Oliveira 2006*a*). □

Lemma C.18. *(*) STOP □ A = A*

Proof The proof lies on (Oliveira 2006*a*). □

Law C.2.16. ***Sequence, External and Internal Choice Process to Basic Process**:*
For op $\in \{ \; ; \; , \; \Box, \sqcap \}$
P op Q = **begin state** *State* $\widehat{=}$
P.State $\wedge$ *Q.State*
P.PPar $\wedge_{\Xi}$ *Q.State*
Q.PPar $\wedge_{\Xi}$ *P.State*
• *P.Act op Q.Act*
end

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.17. ***Parallel and Interleave Processes to Basic Process**:*
For op = $[\![CS]\!]$
P $[\![cs]\!]$ *Q* = **begin state** *State* $\widehat{=}$
P.State $\wedge$ *Q.State*
P.PPar $\wedge_{\Xi}$ *Q.State*
Q.PPar $\wedge_{\Xi}$ *P.State*
• *P.Act* $[\![\alpha(P.State) \mid cs \mid \alpha(Q.State)]\!]$ *Q.Act*
end

Proof The law is referenced on document (Oliveira 2006*a*). □

Law C.2.18. ***Rename Basic Process to Basic Process with Rename Main Action**:*
*(***begin** *PARS • A* **end***) [a1, ... an := b1, ..., bn]*
*= (***begin** *PARS • A [a1, ... an := b1, ..., bn]* **end***)*

Proof The proof lies on (Oliveira 2006*a*) □

Law C.2.19. *(*)* **begin** *PPars • A* **end** $\setminus$ *CS* = **begin** *PPars • (A* $\setminus$ *CS)* **end**

The law is referenced on document (Oliveira 2006*a*).

Law C.2.20.

***Internal Choice Refinement Monotonic**:*
$A_1 \sqcap A_2 \sqsubseteq B_1 \sqcap B_2$ *provided* $A_1 \sqsubseteq B_1$ *and* $A_2 \sqsubseteq B_2$

Proof Proof lies on document (Oliveira 2006*a*). □

Law C.2.21.

***Parallelism Refinement Monotonic**:*
$A_1 \ [\![CS]\!] \ A_2 \sqsubseteq B_1 \ [\![CS]\!] \ B_2$ *provided* $A_1 \sqsubseteq B_1$ *and* $A_2 \sqsubseteq B_2$

Proof Proof lies on document (Oliveira 2006*a*). □

Law C.2.22.

***External Choice Refinement Monotonic**:*
$A_1 \ \Box \ A_2 \sqsubseteq B_1 \ \Box \ B_2$ *provided* $A_1 \sqsubseteq B_1$ *and* $A_2 \sqsubseteq B_2$

Proof Proof lies on document (Oliveira 2006*a*). □

Law C.2.23. ***R2 Conjunction Not Mentioning Trace*** *$p \wedge R2\ (P) = R2\ (p \wedge P)$, provided that p does not mention tr and tr'*

The proof lies on the documents (Oliveira 2006*a*).

Cavalcanti and Woodcock's Laws

Law C.2.24. ***External Choice Idempotence**:*
$(P \ \Box \ P) = P$

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)

□

Law C.2.25. ***Refinement of Non-Determinism**:*
$A \sqcap B \sqsubseteq A$

Proof The proof lies on document (Cavalcanti & Woodcock 2013)

□

Law C.2.26. ***Lift is CSP4**: Lift (v := e) ; Skip = Lift (v := e)*

Proof The proof lies on the document (Cavalcanti & Woodcock 2013). All definitions, lemmas, theorems, laws and rules from CML can be used for *Circus* (Oliveira, Sampaio, Antonino, Ramos, Cavalcanti & Woodcock 2013).
□

Law C.2.27. ***Lift Composition**: Lift (s ; v := w_1) = Lift (s) ; Lift (v := w_1)*

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)
□

Law C.2.28. ***Lift External Choice**: Lift(s) ; (P □ Q) = (Lift(s) ; P) □ (Lift(s) ; Q)*

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)
□

Law C.2.29.

***Lift Left Unit**: Lift (s) ; Lift (t) = Lift (t)*

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)
□

Law C.2.30. ***Lift Leading Substitution**:*
Lift (s) ; P = Lift (s) ; P [e / w_1], provided that (s ; (w_1 = e))

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)
□

Law C.2.31. *(*) Lift (s) ; Lift (v := e) = Lift (s) ; v :=$_{RD}$ e*

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)
□

Law C.2.32.

***Substitution**:*
Lift (v := w) [e / w] = Lift (v := e)

Proof The proof lies on the document (Cavalcanti & Woodcock 2013)

□

Law C.2.33. *(*) Lift (var x := w_1) = var x ; Lift (x := w_1)*

Proof The proof lies on document (Cavalcanti & Woodcock 2013)

□

Law C.2.34. ***Reactive Design Assignment Declaration**:*
$var\ x\ ;\ x :=_{RD} w = var\ x :=_{RD} w$

Proof The proof lies on document (Cavalcanti & Woodcock 2013)

□

Law C.2.35. *(*)* $var_{RD}\ x :=_{RD} w\ ;\ P = P\ (w \vee x)$

Proof The proof lies on document (Cavalcanti & Woodcock 2013)

□

Law C.2.36. ***Input Absorption**:*
$Lift(s)\ ;\ d?x : T \to A = (Lift(s)\ ;\ d.w \to A\ [\ w\ /\ x\ ]) \Box (Lift(s)\ ;\ d?x : T \to A)$

Proof The proof lies on document (Cavalcanti & Woodcock 2013)
□

Law C.2.37. ***Parallel Distributivity**:*
$Lift\ (s)\ ;\ (P \ [\![x_1 \mid cs \mid x_2]\!]\ Q) = (Lift(s)\ ;\ P) \ [\![x_1 \mid cs \mid x_2]\!]\ (Lift(s)\ ;\ Q)$

Proof The proof lies on document (Cavalcanti & Woodcock 2013)
□

Law C.2.38. ***Lift Semi-Idempotence**:*
$s\ ;\ (s \mid x_1\ )_{+x_2} = s$

Proof The proof lies on document (Cavalcanti & Woodcock 2013)
□

Law C.2.39. ***Lift Merge**:*
$Lift\ (s_1) \ [\![x_1 \mid cs \mid x_2]\!]\ Lift\ (s_2) = Lift\ ((s_1 \mid x_1) \wedge (s_2 \mid x_2))$

Proof The law is referenced on document (Cavalcanti & Woodcock 2013) □

Law C.2.40. ***Closure Conjunctive R1** $R1\ (P \wedge Q) = P \wedge Q$, provided that P and Q are R1*

Law C.2.41. ***Closure Conjunctive R3*** *R3 (P ∧ Q) = P ∧ Q, provided that P and Q are R3*

This law was proved on (Cavalcanti & Woodcock 2006).

Law C.2.42. ***Closure Conjunctive R*** *R (P ∧ Q) = P ∧ Q, provided that P and Q are R*

This law was proved on (Cavalcanti & Woodcock 2006).

Law C.2.43. ***Closure Disjunctive R2*** *R2 (P ∨ Q) = P ∨ Q, provided that P and Q are R2*

This law was proved on (Cavalcanti & Woodcock 2006).

Other Laws

On this sub-section we show laws that we created and proved correct.

Law C.2.44. ***Parallel Composition Prefixing Step****:*
(a → A2) ||[ns1 {| cs |} ns2]|| A3 = a → (A2 ||[ns1 {| cs |} ns2]|| A3)

provided that

initials (A3) ⊆ cs
cs ∩ {a} = ∅
wrtV ({a}) ∩ usedV (A3) = ∅
A3 is divergence-free

Proof

(a → A2) ||[ns1 {| cs |} ns2]|| A3
= [Prefixing Sequence theorem: C.2.5]
((a → Skip) ; A2) ||[ns1 {| cs |} ns2]|| A3
= [Parallel Composition Sequence Step C.2.3 and Assumptions]
true
□

Law C.2.45. ***Lifted Assignment Sequenced by end x***
Lift (s ; end x) = Lift (s) ; Lift (end x)

Proof
Lift (s ; end x)
= [Definition of Lift C.24]
R1 ∘ R3 (true ⊢ s ; end x ∧ tr' = tr ∧ ¬ wait')
= [Definition of Sequence C.33]
R1 ∘ R3 (true ⊢ (∃ v0 . s (α, α 0) ∧ end x (α 0, α ')) ∧ tr' = tr ∧ ¬ wait')
= [Predicate Calculus: all variables on v0 are free on R1 and R3]
∃ v0 . R1 ∘ R3 (true ⊢ (s (α, α0) ∧ end x (α0, α')) ∧ tr' = tr ∧ ¬ wait')
= [Predicate Calculus]
∃v0 . R1 ∘ R3 (true ⊢ s (α, α0) ∧ tr' = tr ∧ ¬ wait' ∧ end x (α0, α') ∧ tr' = tr ∧ ¬ wait')
= [Designs And True C.2.46]
∃ v0 .
R1 ∘ R3 (true ⊢ s (α, α0) ∧ tr' = tr ∧ ¬ wait')
∧ R1 ∘ R3 (true ⊢ end x (α0, α') ∧ tr' = tr ∧ ¬ wait')
= [Definition of Lift C.24]
∃ v0 . Lift (s) ∧ Lift (end x)
= [Definition of Sequence C.33, having, from theorem 11 that Lift (s) is a Circus action]
Lift (s) ; Lift (end x)
□

Law C.2.46. ***Designs And True****:*
(true ⊢ Q_1 ∧ Q_2*) = (true* ⊢ Q_1*)* ∧ *(true* ⊢ Q_2*)*

Proof true ⊢ Q1 ∧ Q2
= [Predicate Calculus]
true ∧ ok ⇒ (Q1 ∧ Q2 ∧ ok')
= [Predicate Calculus]
ok ⇒ (Q1 ∧ Q2 ∧ ok')
= [Predicate Calculus]
(¬ok) ∨ (Q1 ∧ Q2 ∧ ok')
= [Predicate Calculus]
(¬ok ∨ Q1) ∧ (¬ok ∨ Q2) ∧ (¬ok ∨ ok')
= [Predicate Calculus]
((¬ok ∨ Q1) ∧ (¬ok ∨ ok')) ∧ ((¬ok ∨ Q2) ∧ (¬ok ∨ ok'))
= [Predicate Calculus]
(¬ok ∨ (Q1 ∧ ok')) ∧ (¬ok ∨ (Q2 ∧ ok'))

= [Predicate Calculus]
(ok ⇒ Q1 ∧ ok') ∧ (ok ⇒ Q2 ∧ ok')
= [Predicate Calculus]
(true ∧ ok ⇒ Q1 ∧ ok') ∧ (true ∧ ok ⇒ Q2 ∧ ok')
= [Predicate Calculus]
(true ⊢ Q1) ∧ (true ⊢ Q2)
□

Law C.2.47. *P ◁ b ▷ (Q ∧ R) = (P ◁ b ▷ Q) ∧ (P ◁ b ▷ R)*

Proof P ◁ b ▷ (Q ∧ R)
= [Definition of Conditional]
(b ∧ P) ∨ (¬ b ∧ Q ∧ R)
= [Predicate Calculus]
((b ∧ P) ∨ (¬ b ∧ Q ∧ ¬ b ∧ R))
= [Predicate Calculus]
((b ∧ P) ∨ (¬ b ∧ Q)) ∧ ((b ∧ P) ∨ (¬ b ∧ R))
= [Definition of Conditional]
(P ◁ b ▷ Q) ∧ (P ◁ b ▷ R)
□

Law C.2.48. ***Lift And**:*
Lift (S ∧ P) = Lift (S) ∧ Lift (P)

Proof LHS
= Lift (S ∧ P)
= [Definition of Lift (C.24)]
R1 (R3 (true ⊢ S ∧ P ∧ tr' = tr ∧ ¬ wait'))
= [Predicate Calculus]
R1 (R3 (true ⊢ S ∧ tr' = tr ∧ ¬ wait' ∧ P ∧ tr' = tr ∧ ¬ wait'))
= [Predicate Calculus]
R1 (R3 (true ⊢ (S ∧ tr' = tr ∧ ¬ wait') ∧ (P ∧ tr' = tr ∧ ¬ wait')))
= [Designs And True C.2.46]
R1 ∘ R3 ((true ⊢ (S ∧ tr' = tr ∧ ¬ wait')) ∧ (true ⊢ (P ∧ tr' = tr ∧ ¬ wait')))
= [Closure Conjunctive R1 (C.2.40) and Closure Conjunctive R3 (C.2.41)]
R1 ∘ R3 (true ⊢ (S ∧ tr' = tr ∧ ¬ wait')) ∧ R1 ∘ R3 (true ⊢ (P ∧ tr' = tr ∧ ¬ wait'))
= [Definition of Lift (C.24)]
Lift (S) ∧ Lift (P)
= RHS □

Law C.2.49. ***R2 and R3 Composition Commutative:***
R2 ∘ R3 (P) = R3 ∘ R2 (P)

Proof
R2 ∘ R3 (P)
= [Definition of R3 C.21]
R2 ($II_{rea} \lhd$ wait $\rhd$ P)
= [Definition of Conditional C.18]
R2 ((wait $\wedge$ II_{rea}) $\vee$ ($\neg$ wait) $\wedge$ P))
= [Disjunctive Closure R2 C.2.43]
R2 (wait $\wedge$ II_{rea}) $\vee$ R2 ($\neg$ wait) $\wedge$ P)
= [R2 Conjunctive Not Mentioning Trace C.2.23 (having that wait and $\neg$ wait do not mention tr and tr')]
(wait $\wedge$ R2 (II_{rea})) $\vee$ ($\neg$ wait) $\wedge$ R2 (P))
= [Definition of Conditional C.18]
(R2 (II_{rea})) $\lhd$ wait $\rhd$ (R2 (P))
= [Definition of R3 C.21]
R3 ∘ R2 (P)
□

C.3 Proof of Soundness for rules

C.3.1 Assignment

Attached Rule 1.

Assignment:

$$\frac{c \qquad (s\ ;\ (w0 = e))}{(c \mid s \models v := e) \xrightarrow{\tau} (c \wedge (s\ ;\ (w0 = e)) \mid s\ ;\ v := w0 \models Skip)}$$

The proof, on document (Cavalcanti & Woodcock 2013), consists on proving that the LHS of the refinement expression equals the RHS.

Proof
RHS
=
Lift (s ; v := w0) ; Skip
= **[Law LiftCSP4 C.2.26]**
Lift (s ; v := w0)
= **[Law Lift Composition C.2.27]**
Lift (s) ; Lift (v := w0)
= **[Law Lift Leading Substitution** C.2.30]
Lift (s) ; Lift (v := w0) [e/w0]
= **[Law Substitution C.2.32]**
Lift (s) ; Lift (v := e)
= **[Reactive Design Assignment C.17]**
Lift (s) ; v :=$_{RD}$ e
=
LHS □

C.3.2 Prefixing*

Attached Rule 2.

Input*:*

$$\frac{c \quad T \neq \emptyset \quad x \notin \alpha(s)}{(c \mid s \models d?x{:}T \to A) \xrightarrow{d.w_0} (c \wedge w_0 \in T \mid s \,;\, \textbf{var}\ x := w_0 \models \textit{let}\ x \bullet A)}$$

Proof

(c | s $\models$ d?x:T $\to$ A) $\xrightarrow{d.w_0}$ (c $\wedge$ w_0 $\in$ T | s ; **var** x := w_0 $\models$ **let** x $\bullet$ A)

= **[Definition C.26]**

$\forall$ w . c $\wedge$ w_0 : T $\Rightarrow$

$\big($ Lift(s); d?x: T $\to$ A $\sqsubseteq$ (Lift(s; **var** x := w_0); $d._1$ $\to$ **let** x $\bullet$ A) $\Box$ (Lift(s); d?x: T $\to$ A) $\big)$

On Deliverable (Cavalcanti & Woodcock 2013), the proof consists on proving that the RHS (Right-Hand Side) equals the LHS (Left-Hand Side):

RHS

=

(Lift(s; **var** x := w_0); $d._1$ $\to$ **let** x $\bullet$ A) $\Box$ (Lift(s); d?x: T $\to$ A)

= **[Lemma C.28]**

(Lift (s ; **var** x := w_0) ; d.w_0 $\to$ A) $\Box$ (Lift(s) ; d?x : T $\to$ A)

= **[Law Lift Composition C.2.27]**

(Lift(s); Lift(**var** x := w_0); d.w_0 $\to$ A) $\Box$ (Lift(s); d?x : T $\to$ A)

= **[Law Lift Var C.2.33]**

$\Big($ $(Lift(s);$ **var**$x;$ $Lift(x := w_0);$ $d.w_0 \to A)$ $\Box$ $(Lift(s);$ $d?x : T \to A)$ $\Big)$

= **[Reactive Design Assignment C.17]**

(Lift(s); **var** x; x :=$_{RD}$ w_0; d.w_0 $\to$ A) $\Box$ (Lift(s); d?x : T $\to$ A)

= **[Reactive Design Declaration C.2.34]**

(Lift(s); **var** x :=$_{RD}$ w_0; d.w_0 $\to$ A) $\Box$ (Lift(s); d?x : T $\to$ A)

= **[Reactive Design Declaration Elimination C.2.35]**

(Lift(s); (d.w_0 $\to$ A) (w_0/x)) $\Box$ (Lift(s) ; d?x : T $\to$ A)

= **[Substitution C.2.32]**

(Lift(s); d.w_0 $\to$ A(w_0/x)) $\Box$ (Lift(s); d?x : T $\to$ A)

= **[Input Absorption C.2.36]**

(Lift(s) ; d?x : T $\to$ A)

=
LHS □

Attached Rule 3.

Output*:

$$\frac{c \quad s\ ;\ (w_0 = e)}{(c \mid s \models d!e \to A) \xrightarrow{d.w_0} (c \wedge s\ ;\ (w_0 = e) \mid s \models A)}$$

Proof

$(c \mid s \models d!e \to A) \xrightarrow{d.w_0} (c \wedge s\ ;\ (w_0 = e) \mid s \models A)$

= **[Definition C.26]**

$\forall\, w \,.\, c \wedge (c \wedge s\ ;\ (w_0 = e)) \Rightarrow$

$\big(\text{Lift}\ (s)\ ;\ d.e \to A \sqsubseteq (\text{Lift}\ (s)\ ;\ d.w_0 \to A) \,\Box\, (\text{Lift}\ (s)\ ;\ d.e \to A)\big)$

= **[Lift Leading Substitution C.2.30] and assumption (s ; (w_0 = e))**

$\forall\, w \,.\, c \wedge (c \wedge s\ ;\ (w_0 = e)) \Rightarrow$

$\big(\text{Lift}\ (s)\ ;\ d.e \to A \sqsubseteq (\text{Lift}\ (s)\ ;\ (d.w_0 \to A)\ [e\ /\ w_0]) \,\Box\, (\text{Lift}\ (s)\ ;\ d.e \to A)\big)$

= **[Substitution C.2.32] and assumption (s ; (w_0 = e))**

$\forall\, w \,.\, c \wedge (c \wedge s\ ;\ (w_0 = e)) \Rightarrow$

$\big(\text{Lift}\ (s)\ ;\ d.e \to A \sqsubseteq (\text{Lift}\ (s)\ ;\ d.e \to A\ [e\ /\ w_0]) \,\Box\, (\text{Lift}\ (s)\ ;\ d.e \to A)\big)$

= **[Substitution C.2.32] and w_0 does not occur in A**

$\forall\, w \,.\, c \wedge (c \wedge s\ ;\ (w_0 = e)) \Rightarrow$

$\big(\text{Lift}\ (s)\ ;\ d.e \to A \sqsubseteq (\text{Lift}\ (s)\ ;\ d.e \to A) \,\Box\, (\text{Lift}\ (s)\ ;\ d.e \to A)\big)$

= **[External Choice Idempotence C.2.24]**

$\forall\, w \,.\, c \wedge (c \wedge s\ ;\ (w_0 = e)) \Rightarrow \big(\text{Lift}\ (s)\ ;\ d.e \to A \sqsubseteq \text{Lift}\ (s)\ ;\ d.e \to A\big)$

= **[Refinement Equal Sides 1]**

$\forall\, w \,.\, c \wedge (c \wedge s\ ;\ (w_0 = e)) \Rightarrow \text{true}$

= **[Predicate Calculus]**

$\forall\, w \,.\, \text{true}$

= **true** □

C.3.3 Variable Block

All laws for Variable Block (Variable Block Begin, Variable Block Visible and Variable Block End) were proved by Barrocas, on this Thesis.

Attached Rule 4.

Variable Block Begin:

$(c \mid s \models \textbf{var}\ x : T \bullet A) \xrightarrow{\tau} (c \wedge w_0 \in T \mid s\ ;\ \textbf{var}\ x := w_0 \models \textbf{let}\ x \bullet A)$

Proof

(c ⌈ s ⊨ **var** x : T • A) $\xrightarrow{\tau}$ (c ∧ w_0 ∈ T ⌈ s ; **var** x := w_0 ⊨ **let** x • A)

= [Definition of Silent Transition C.25]

∀w . (c ∧ c ∧ w_0 : T) ⇒ (Lift (s) ; **var** x : T • A) ⊑ (Lift (s ; **var** x := w_0) ; let x • A)

= [w is universally quantified, so we abstract it]

(c ∧ c ∧ w_0 : T) ⇒ (Lift (s) ; **var** x : T • A) ⊑ (Lift (s ; **var** x := w_0) ; let x • A)

= [Lift Composition C.2.27]

(c ∧ c ∧ w_0 : T) ⇒ (Lift (s) ; **var** x : T • A) ⊑ (Lift (s) ; Lift (**var** x := w_0) ; let x • A)

= [Lift Left Unit C.2.29]

(c ∧ c ∧ w_0 : T) ⇒ (**var** x : T • A) ⊑ (Lift (**var** x := w_0) ; let x • A)

= [(x := e) = (var x ; x := e)]

(c ∧ c ∧ w_0 : T) ⇒ (**var** x : T • A) ⊑ (Lift (**var** x; x := w_0) ; let x • A)

= [Let definition C.28]

(c ∧ c ∧ w_0 : T) ⇒ (**var** x : T • A) ⊑ (Lift (**var** x; x := w_0) ; A)

= [Refinement Definition C.14]

(c ∧ c ∧ w_0 : T) ⇒ [(Lift (**var** x; x := w_0) ; A) ⇒ (**var** x : T • A)]

= [Lemma C.8 and assms (w_0 : T)]

(c ∧ c ∧ w_0 : T) ⇒ [true]

= [Predicate Calculus]

(c ∧ c ∧ w_0 : T) ⇒ true

= [Predicate Calculus]

true

□

Attached Rule 5.

Variable Block Visible:

$$\frac{(c_1 \mid s_1 \models A_1\ ) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \textbf{let}\ x \bullet A_1\ ) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{let}\ x \bullet A_2)}$$

Proof

$(c_1 \mid s_1 \models \textbf{let}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{let}\ x \bullet A_2)$

= [Definition of let C.28]

$(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{let}\ x \bullet A_2)$

= [Definition of let C.28]

$(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)$

= [Assumption $(\mathbf{c_1 \mid s_1 \models A_1}) \xrightarrow{l} (\mathbf{c_2 \mid s_2 \models A_2})$]

true

□

Attached Rule 6.

***Variable Block End**:*

$$\frac{c}{(c \mid s \models \textbf{\textit{let}}\ x \bullet Skip) \xrightarrow{l} (c \mid s\ ;\ \textbf{\textit{end}}\ x \models Skip)}$$

Proof

$\forall w.(c \wedge c) \Rightarrow (Lift(s);\ let\,x \bullet Skip) \sqsubseteq (Lift(s;\ end\,x);\ Skip)$

= [w is universally quantified, so we abstract it]

$(c \wedge c) \Rightarrow (Lift(s);\ let\,x \bullet Skip) \sqsubseteq (Lift(s;\ end\,x);\ Skip)$

= [Assumption c]

$(Lift(s);\ let\,x \bullet Skip) \sqsubseteq (Lift(s;\ end\,x);\ Skip)$

= [Definition of let [C.28]]

$(Lift(s);\ Skip) \sqsubseteq (Lift(s;\ end\,x);\ Skip)$

= [Lifted Assignment Sequenced by end x C.2.45]

$(Lift(s);\ Skip) \sqsubseteq (Lift(s);\ Lift(end\,x);\ Skip)$

As Sequence is monotonic with respect to refinement and Lift (s) is refined by itself, if we prove that Skip ⊑ (Lift (end x) ; Skip), we also prove the above expression:

$Skip \sqsubseteq (Lift(end\,x);\ Skip)$

= [Sequence Denotational Definition C.33]

$Skip \sqsubseteq \exists\, v0.(Lift(end\,x)[v, v0] \wedge Skip[v0, v'])$

= [end x definition]

$Skip \sqsubseteq \exists v0.(Lift(true \wedge \exists x')[v, v0] \wedge Skip[v0, v'])$

= [Lift Definition C.24 and true ∧ ∃ x' is R2]

$Skip \sqsubseteq \exists v0.(R(true \vdash true \wedge \exists x0.tr0 = tr \wedge \neg\, wait0) \wedge Skip[v0, v'])$

= [Skip Denotational Definition]

$$Skip \sqsubseteq \exists v0. \left(\begin{array}{l} R(true \vdash true \wedge \exists x0.tr0 = tr \wedge \neg\, wait0) \\ \wedge\, (R(true \vdash tr' = tr0 \wedge \neg\, wait' \wedge v' = v0)) \end{array} \right)$$

= [Predicate Calculus]

$$Skip \sqsubseteq \exists\, v0\,.\, \left(\begin{array}{l} R\,(true \vdash true \wedge tr0 = tr \wedge \neg\, wait0) \\ \wedge\, (R(true \vdash tr' = tr0 \wedge \neg\, wait' \wedge v' = v0)) \end{array} \right)$$

= [Skip definition]

$$R(true \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v) \sqsubseteq \exists\, v0\,.\, \left(\begin{array}{l} R\,(true \vdash true \wedge tr0 = tr \wedge \neg\, wait0) \\ \wedge\, (R(true \vdash tr' = tr0 \wedge \neg\, wait' \wedge v' = v0)) \end{array} \right)$$

= [Designs And True C.2.46 and Predicate Calculus]

$$R(true \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v) \sqsubseteq \exists\, v0\,.\, \left(R\,(true \vdash tr' = tr \wedge \neg\, wait0 \wedge \neg\, wait' \wedge v' = v0) \right)$$

= [Predicate Calculus]

$$R(true \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v) \sqsubseteq \exists\, v\,.\, \left(R\,(true \vdash tr' = tr \wedge \neg\, wait0 \wedge \neg\, wait' \wedge v' = v) \right)$$

= [Factor $\neg$ wait0 makes the right side stronger than the left side, thus it is a refinement]

= true

□

C.3.4 Sequence

Attached Rule 7.

Sequence Progress:

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1 \; ; B) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \; ; B)}$$

Proof

We will divide the proof in two cases, both of which have c_1 and c_2 being true. In the case that some of them is false, the antecedent of the implication is false, what makes the expression directly true.

Case 1 : l = τ and c_1 and c_2 are true

The assumption

$(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)$

equals (by the definition of Silent Transition C.25)

$\forall$ w . $c_1 \wedge c_2 \Rightarrow$ Lift (s_1) ; $A_1 \sqsubseteq$ Lift (s_2) ; A_2

Abstracting w, as it is universally quantified, we have

$c_1 \wedge c_2 \Rightarrow$ Lift (s_1) ; $A_1 \sqsubseteq$ Lift (s_2) ; A_2

Having c_1 and c_2 being true, by predicate calculus, we have:

[Derived Assumption]: Lift (s_1) ; $A_1 \sqsubseteq$ Lift (s_2) ; A_2

Now we start proving:

$(c_1 \mid s_1 \models A_1 \; ; B) \xrightarrow{\tau} (c_2 \mid s_2 \models A_2 \; ; B)$
= [Definition C.25, of τ transition]
$\forall$ w . $c_1 \wedge c_2 \Rightarrow$ Lift (s_1) ; A_1 ; B $\sqsubseteq$ Lift (s_2) ; A_2 ; B

= [w is universally quantified, so we abstract it]
$c_1 \wedge c_2 \Rightarrow \text{Lift } (s_1) \text{ ; } A_1 \text{ ; } B \sqsubseteq \text{Lift } (s_2) \text{ ; } A_2 \text{ ; } B$
= [Derived Assumption with Monotonicity of Refinement (6) lead us to: Lift (s_1) ; A_1 ; B $\sqsubseteq$ Lift (s_2) ; A_2 ; B]
= true

Case 2 : l is not silent ($\neq \tau$) and c1 and c2 are true
The assumption

$$(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)$$

equals (by the definition of Labelled Transition C.26)

$\forall$ w . $c_1 \wedge c_2 \Rightarrow \text{Lift } (s_1) \text{ ; } A_1 \sqsubseteq (\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2) \square (\text{Lift } (s_1) \text{ ; } A_1)$
= [w is universally quantified, so we abstract it]
$c_1 \wedge c_2 \Rightarrow \text{Lift } (s_1) \text{ ; } A_1 \sqsubseteq (\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2) \square (\text{Lift } (s_1) \text{ ; } A_1)$

Having c_1 and c_2 being true, by predicate calculus, we have:

$\forall$ w . $\text{Lift } (s_1) \text{ ; } A_1 \sqsubseteq (\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2) \square (\text{Lift } (s_1) \text{ ; } A_1)$

and this lead us to

[Derived Assumption 2]:
$\text{Lift } (s_1) \text{ ; } A_1 \sqsubseteq (\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2) \square (\text{Lift } (s_1) \text{ ; } A_1)$

Expression to prove:
$$(c_1 \mid s_1 \models A_1 \text{ ; } B) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \text{ ; } B) =$$
$\forall$ w . $c_1 \wedge c_2 \Rightarrow (\text{Lift } (s_1) \text{ ; } A_1 \text{ ; } B) \sqsubseteq (\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2 \text{ ; } B) \square (\text{Lift } (s_1) \text{ ; } A_1 \text{ ; } B)$
= [w is universally quantified, so we abstract it]
$c_1 \wedge c_2 \Rightarrow (\text{Lift } (s_1) \text{ ; } A_1 \text{ ; } B) \sqsubseteq (\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2 \text{ ; } B) \square (\text{Lift } (s_1) \text{ ; } A_1 \text{ ; } B)$
= [External Choice/Sequence Distributive C.2.11]
$c_1 \wedge c_2 \Rightarrow (\text{Lift } (s_1) \text{ ; } A_1 \text{ ; } B) \sqsubseteq ((\text{Lift } (s_2) \text{ ; c.w1} \rightarrow A_2) \square (\text{Lift } (s_1) \text{ ; } A_1) \text{ ; } B)$
= [Derived Assumption 2 with Monotonicity of Refinement (6) lead us to: (Lift (s_1) ; A_1 ; B) $\sqsubseteq$ (Lift (s_2) ; c.w1 $\rightarrow$ A_2 ; B) $\square$ (Lift (s_1) ; A_1 ; B)]
$c_1 \wedge c_2 \Rightarrow$ true

= [(P $\Rightarrow$ true) = true]
= true

□

Attached Rule 8.

***Sequence End**:*

$$\frac{}{(c \mid s \models \mathit{Skip} \;;\; A) \xrightarrow{\tau} (c \mid s \models A)}\; c$$

Proof $(c \mid s \models \mathit{Skip} \;;\; A) \xrightarrow{\tau} (c \mid s \models A)$
= [Theorem C.2.2]
$(c \mid s \models A) \xrightarrow{\tau} (c \mid s \models A)$
= [Theorem 2]
true □

C.3.5 Internal Choice*

Attached Rule 9.

Internal Choice Left*:*
$(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\tau} (c \mid s \models A_1)$

Proof
$(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\tau} (c \mid s \models A_1)$
= [Definition of Silent Transition C.25]
$\forall\, w \,.\, c \wedge c \Rightarrow \text{Lift}\ (s)\ ;\ (A_1 \sqcap A_2) \sqsubseteq \text{Lift}\ (s)\ ;\ A_1$
= [w is universally quantified, so we abstract it]
$c \wedge c \Rightarrow \text{Lift}\ (s)\ ;\ (A_1 \sqcap A_2) \sqsubseteq \text{Lift}\ (s)\ ;\ A_1$
= [Refinement of Non-Determinism C.2.25]
$c \wedge c \Rightarrow \text{Lift}\ (s)\ ;\ A_1 \sqsubseteq \text{Lift}\ (s)\ ;\ A_1$
= [Refinement Equal sides 1]
$c \wedge c \Rightarrow \text{true}$
= [Predicate Calculus]
= true □

Attached Rule 10.

Internal Choice Right*:*
$(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\tau} (c \mid s \models A_2)$

Proof
$(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\tau} (c \mid s \models A_2)$
= Definition of Silent Transition C.25
$\forall\, w \,.\, c \wedge c \Rightarrow \text{Lift}\ (s)\ ;\ (A_1 \sqcap A_2) \sqsubseteq \text{Lift}\ (s)\ ;\ A_2$
= [w is universally quantified, so we abstract it]
$c \wedge c \Rightarrow \text{Lift}\ (s)\ ;\ (A_1 \sqcap A_2) \sqsubseteq \text{Lift}\ (s)\ ;\ A_2$
= [Refinement Non-Determinism C.2.25]
$c \wedge c \Rightarrow \text{Lift}\ (s)\ ;\ A_1 \sqsubseteq \text{Lift}\ (s)\ ;\ A_2$
= [Refinement Equal sides 1]
$c \wedge c \Rightarrow \text{true}$
= [Predicate Calculus]
= true □

C.3.6 Guard

Attached Rule 11.

***Guard**:*

$$\frac{c \qquad (s\ ;\ g)}{(c \mid s \models g\ \&\ A) \xrightarrow{\tau} (c \wedge (s\ ;\ g) \mid s \models A)}$$

Proof

[Case 1: g is true]

$(c \mid s \models g\ \&\ A) \xrightarrow{\tau} (c \wedge (s\ ;\ g) \mid s \models A)$

= [Definition of Silent Transition C.25]

$\forall\ w\ .\ c \wedge c \wedge s\ ;\ g \Rightarrow (\text{Lift}\ (s)\ ;\ g\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [w is universally quantified, so we abstract it]

$c \wedge c \wedge s\ ;\ g \Rightarrow (\text{Lift}\ (s)\ ;\ g\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [Assumptions c and s ; g]

$(\text{true} \wedge \text{true} \wedge \text{true}) \Rightarrow (\text{Lift}\ (s)\ ;\ g\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [Predicate Calculus]

$(\text{Lift}\ (s)\ ;\ g\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

[True guard C.2.4]

$(\text{Lift}\ (s)\ ;\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [Refinement Equal Sides 1]

true

[Case 2: g is false]

$(c \mid s \models g\ \&\ A) \xrightarrow{\tau} (c \wedge (s\ ;\ g) \mid s \models A)$

= [Definition of Silent Transition C.25]

$\forall\ w\ .\ c \wedge c \wedge s\ ;\ g \Rightarrow (\text{Lift}\ (s)\ ;\ g\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [w is universally quantified, so we abstract it]

$c \wedge c \wedge s\ ;\ g \Rightarrow (\text{Lift}\ (s)\ ;\ g\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [Case 2: g is false (replacing g by false)]

$c \wedge c \wedge s\ ;\ \text{false} \Rightarrow (\text{Lift}\ (s)\ ;\ \text{false}\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [S Sequence False Equals False 5]

$c \wedge c \wedge \text{false} \Rightarrow (\text{Lift}\ (s)\ ;\ \text{false}\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [Predicate Calculus]

$\text{false} \Rightarrow (\text{Lift}\ (s)\ ;\ \text{false}\ \&\ A) \sqsubseteq (\text{Lift}\ (s)\ ;\ A)$

= [Predicate Calculus]

true

□

C.3.7 External Choice*

Attached Rule 12.

External Choice Begin*:*
$(c \mid s \models A_1 \,\Box\, A_2) \xrightarrow{\tau} (c \mid s \models (loc\ s \bullet A_1) \boxplus (loc\ s \bullet A_2))$

Proof
$\forall$ w . c $\wedge$ c $\Rightarrow$ (Lift (s) ; $A_1 \,\Box\, A_2 \sqsubseteq$ (loc s $\bullet$ A_1) $\boxplus$ (loc s $\bullet$ A_2))
= [Definition of Extra Choice C.30]
$\forall$ w . c $\wedge$ c $\Rightarrow$ (Lift (s) ; $A_1 \,\Box\, A_2 \sqsubseteq$ (loc s $\bullet$ A_1) $\Box$ (loc s $\bullet$ A_2))
= [w is universally quantified, so we abstract it]
c $\wedge$ c $\Rightarrow$ (Lift (s) ; $A_1 \,\Box\, A_2 \sqsubseteq$ (loc s $\bullet$ A_1) $\Box$ (loc s $\bullet$ A_2))
= [Definition of loc C.29]
c $\wedge$ c $\Rightarrow$ (Lift (s) ; $A_1 \,\Box\, A_2 \sqsubseteq$ (Lift (s) ; A_1) $\Box$ (Lift (s) ; A_2))
= [Lift External Choice C.2.28]
c $\wedge$ c $\Rightarrow$ (Lift (s) ; $A_1 \,\Box\, A_2 \sqsubseteq$ (Lift (s) ; Lift (s) ; $A_1 \,\Box\, A_2$))
= [Lift Left Unit C.2.29]
c $\wedge$ c $\Rightarrow$ (Lift (s) ; $A_1 \,\Box\, A_2 \sqsubseteq$ (Lift (s) ; ($A_1 \,\Box\, A_2$)))
= [Parenthesis]
c $\wedge$ c $\Rightarrow$ (Lift (s) ; ($A_1 \,\Box\, A_2$) $\sqsubseteq$ (Lift (s) ; ($A_1 \,\Box\, A_2$)))
= [Refinement Equal Sides 1]
c $\wedge$ c $\Rightarrow$ true
= [(P $\Rightarrow$ true) = true]
true
□

Attached Rule 13.

External Choice Skip*:*
$(c \mid s \models loc\ s_1 \bullet Skip \boxplus loc\ s_2 \bullet A) \xrightarrow{\tau} (c \mid s \models Skip)$

Proof (c | s $\models$ loc s_1 $\bullet$ Skip $\boxplus$ loc s_2 $\bullet$ A) $\xrightarrow{\tau}$ (c | s_1 $\models$ Skip)
= [Definition of Silent Transition C.25]
$\forall$ w . c $\Rightarrow$ Lift (s_1) ; (loc s_1 $\bullet$ Skip $\boxplus$ loc s_2 $\bullet$ A) $\sqsubseteq$ Lift (s_1) ; Skip
= [w is universally quantified, so we abstract it]
c $\Rightarrow$ Lift (s_1) ; (loc s_1 $\bullet$ Skip $\boxplus$ loc s_2 $\bullet$ A) $\sqsubseteq$ Lift (s_1) ; Skip
= [Definition of Loc C.29]

$c \Rightarrow \text{Lift}\ (s_1)\ ;\ (\text{Lift}\ (s_1)\ ;\ \text{Skip} \boxplus \text{Lift}\ (s_2)\ ;\ A) \sqsubseteq \text{Lift}\ (s_1)\ ;\ \text{Skip}$

= [Lift is CSP4 C.2.26]

$c \Rightarrow \text{Lift}\ (s_1)\ ;\ (\text{Lift}\ (s_1) \boxplus \text{Lift}\ (s_2)\ ;\ A) \sqsubseteq \text{Lift}\ (s_1)\ ;\ \text{Skip}$

= [Extra Choice Definition C.30]

$c \Rightarrow \text{Lift}\ (s_1)\ ;\ (\text{Lift}\ (s_1) \Box \text{Lift}\ (s_2)\ ;\ A) \sqsubseteq \text{Lift}\ (s_1)\ ;\ \text{Skip}$

= [External Choice Assignment]

$c \Rightarrow \text{Lift}\ (s_1)\ ;\ \text{Lift}\ (s_1) \sqsubseteq \text{Lift}\ (s_1)\ ;\ \text{Skip}$

= [Lift is CSP4 C.2.26]

$c \Rightarrow \text{Lift}\ (s_1)\ ;\ \text{Lift}\ (s_1) \sqsubseteq \text{Lift}\ (s_1)$

= [Lift Left Unit C.2.29]

$c \Rightarrow \text{Lift}\ (s_1) \sqsubseteq \text{Lift}\ (s_1)$

= [Refinement Equal Sides 1]

$c \Rightarrow \text{true}$

= [Predicate Calculus]

= true

□

Attached Rule 14.

External Choice End*:

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3)}{(c \mid s \models \mathit{loc}\ s_1 \bullet A_1 \boxplus \mathit{loc}\ s_2 \bullet A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3)}$$

Proof

$(c \mid s \models \text{loc}\ s_1 \bullet A_1 \boxplus \text{loc}\ s_2 \bullet A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3)$

= [Definition of labelled transition C.26]

$\forall\ w\ .\ c \wedge c_3 \Rightarrow \text{Lift}\ (s)\ ;\ ((\text{loc}\ s_1 \bullet A_1) \boxplus (\text{loc}\ s_2 \bullet A_2)) \sqsubseteq (\text{Lift}\ (s_3)\ ;\ l \rightarrow A_3) \Box (\text{Lift}\ (s)\ ;\ ((\ \text{loc}\ s_1 \bullet A_1\ ) \boxplus (\ \text{loc}\ s_2 \bullet A_2)))$

= [w is universally quantified, so we abstract it]

$c \wedge c_3 \Rightarrow \text{Lift}\ (s)\ ;\ ((\text{loc}\ s_1 \bullet A_1) \boxplus (\text{loc}\ s_2 \bullet A_2)) \sqsubseteq (\text{Lift}\ (s_3)\ ;\ l \rightarrow A_3) \Box (\text{Lift}\ (s)\ ;\ ((\ \text{loc}\ s_1 \bullet A_1\ ) \boxplus (\ \text{loc}\ s_2 \bullet A_2)))$

= [Definition of Extra Choice C.30]

$c \wedge c_3 \Rightarrow \text{Lift}\ (s)\ ;\ ((\text{loc}\ s_1 \bullet A_1) \Box (\text{loc}\ s_2 \bullet A_2)) \sqsubseteq (\text{Lift}\ (s_3)\ ;\ l \rightarrow A_3) \Box (\text{Lift}\ (s)\ ;\ ((\ \text{loc}\ s_1 \bullet A_1\ ) \Box (\ \text{loc}\ s_2 \bullet A_2)))$

= [Definition of Loc C.29]

$c \wedge c_3 \Rightarrow \text{Lift}\ (s)\ ;\ ((\text{loc}\ s_1 \bullet A_1) \Box (\text{loc}\ s_2 \bullet A_2)) \sqsubseteq (\text{Lift}\ (s_3)\ ;\ l \rightarrow A_3) \Box (\text{Lift}\ (s)\ ;\ ((\text{Lift}\ (s_1)\ ;\ A_1) \Box (\text{Lift}\ (s_2)\ ;\ A_2)))$

= [Lift External Choice C.2.28]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s_3) ; l $\rightarrow A_3$) $\Box$ (Lift (s) ; Lift (s_1) ; A_1) $\Box$ (Lift (s) ; Lift (s_2) ; A_2)))

= [Lift Left Unit C.2.29]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s_3) ; l $\rightarrow A_3$) $\Box$ (Lift (s_1) ; A_1) $\Box$ (Lift (s_2) ; A_2)))

= [Assumption]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s_1) ; A_1) $\Box$ (Lift (s_2) ; A_2)))

= [Lift Left Unit C.2.29]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s) ; Lift (s_1) ; A_1) $\Box$ (Lift (s) ; Lift (s_2) ; A_2)))

= [Lift External Choice C.2.28]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s) ; (Lift (s_1) ; A_1) $\Box$ Lift (s_2) ; A_2)

= [Definition of Loc C.29]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s) ; (loc $s_1 \bullet A_1$) $\Box$ loc $s_2 \bullet A_2$)

= [Definition of Extra Choice C.30]

$c \wedge c_3 \Rightarrow$ Lift (s) ; ((loc $s_1 \bullet A_1$) $\Box$ (loc $s_2 \bullet A_2$)) $\sqsubseteq$ (Lift (s) ; (loc $s_1 \bullet A_1$) $\boxplus$ loc $s_2 \bullet A_2$)

= [Refinement Equal Sides 1]

$c \wedge c_3 \Rightarrow$ true

= [Predicate Calculus]

= true

$\Box$

Attached Rule 15.

External Choice Silent Left:

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\tau} (c_3 \mid s_3 \models A_3)}{(c \mid s \models loc\ s_1 \bullet A_1 \boxplus loc\ s_2 \bullet A_2) \xrightarrow{\tau} (c \mid s \models loc\ s_3 \bullet A_3 \boxplus loc\ s_2 \bullet A_2)}$$

(c1 | s $\models$ (loc s1 $\bullet$ A1) $\boxplus$ (loc s2 $\bullet$ A2))

$\xrightarrow{\tau}$

(c2 | s $\models$ (loc s3 $\bullet$ A3) $\boxplus$ (loc s2 $\bullet$ A2))

= [loc definition C.29]

(c1 | s $\models$ (Lift (s1); A1) $\boxplus$ (Lift (s2); A2))

$\xrightarrow{\tau}$

(c2 | s $\models$ (Lift (s3); A3) $\boxplus$ (Lift (s2); A2))

= [Extra choice definition C.30]

(c1 ∣ s ⊨ (Lift (s1); A1) □ (Lift (s2); A2))
$\xrightarrow{\tau}$
(c2 ∣ s ⊨ (Lift (s3); A3) □ (Lift (s2); A2))

= [Definition of Silent Transition C.25]

∀w . c1 ∧ c2 ⇒ (Lift (s1); A1) □ (Lift (s2); A2)) ⊑ (Lift (s3); A3) □ (Lift (s2); A2)

= [w is universally quantified, so we abstract it]

c1 ∧ c2 ⇒ (Lift (s1); A1) □ (Lift (s2); A2)) ⊑ (Lift (s3); A3) □ (Lift (s2); A2)

[Assumpt. (Lift (s1) ; A1 ⊑ Lift (s3) ; A3) + Ref.Eq.Sides (1)]

c1 ∧ c2 ⇒ true

= [Predicate Calculus]

true

Attached Rule 16.

***External Choice Silent Right**:*

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{\tau} (c_3 \mid s_3 \models A_3)}{(c \mid s \models loc\ s_1 \bullet A_1 \boxplus loc\ s_2 \bullet A_2) \xrightarrow{\tau} (c \mid s \models loc\ s_1 \bullet A_1 \boxplus loc\ s_3 \bullet A_3)}$$

(c1 ∣ s ⊨ (loc s1 • A1) ⊞ (loc s2 • A2))
$\xrightarrow{\tau}$
(c2 ∣ s ⊨ (loc s1 • A1) ⊞ (loc s3 • A3))

= [loc definition C.29]

(c1 ∣ s ⊨ (Lift (s1); A1) ⊞ (Lift (s2); A2))
$\xrightarrow{\tau}$
(c2 ∣ s ⊨ (Lift (s1); A1) ⊞ (Lift (s3); A3))

= [Extra choice definition C.30]

(c1 ∣ s ⊨ (Lift (s1); A1) □ (Lift (s2); A2))
$\xrightarrow{\tau}$
(c2 ∣ s ⊨ (Lift (s1); A1) □ (Lift (s3); A3))

= [Definition of Silent Transition C.25]

∀w . c2 ∧ c2 ⇒ (Lift (s1); A1) □ (Lift (s2); A2)) ⊑ (Lift (s1); A1) □ (Lift (s3); A3)

= [w is universally quantified, so we abstract it]

c2 ∧ c2 ⇒ (Lift (s1); A1) □ (Lift (s2); A2)) ⊑ (Lift (s1); A1) □ (Lift (s3); A3)

= [Lift (s2) ; A2 ⊑ Lift (s3) ; A3 (this is the assumption) + Refinement Equal Sides 1]

c1 ∧ c2 ⇒ true

= [Predicate Calculus]
true

C.3.8 Parallelism*

Attached Rule 17.

Parallel Begin*:*

$(c \mid s \models A_1 \; [\![s_1 \mid cs \mid s_2]\!] \; A_2)$
$\xrightarrow{\tau}$
$(c \mid s \models (loc\ (s \mid x_1)_{+x2} \bullet A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; (loc\ (s \mid x_2)_{+x1} \bullet A_2))$

Proof

$(c \mid s \models A_1 \; [\![s_1 \mid cs \mid s_2]\!] \; A_2)$
$\xrightarrow{\tau}$
$(c \mid s \models (loc\ (s \mid x_1)_{+x2} \bullet A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; (loc\ (s \mid x_2)_{+x1} \bullet A_2))$
= [Definition of Silent Transition C.25]
$\forall\, w\, .\, c \Rightarrow Lift\ (s)\ ;\ A_1\ |[\ s_1 \mid cs \mid s_2\ ]|\ A_2 \sqsubseteq Lift\ (s)\ ;\ (loc\ (s \mid x_1)_{+x2} \bullet A_1) \; [\![x_1 \mid cs \mid x_2]\!]$
$(loc\ (s \mid x_2)_{+x1} \bullet A_2$
The proof from now on will consist on proving that the RHS of the refinement is equal to the LHS.
RHS =
$Lift\ (s)\ ;\ (loc\ (s \mid x_1)_{+x2} \bullet A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; (loc\ (s \mid x_2)_{+x1} \bullet A_2$
= [Definition of Loc C.29]
$Lift\ (s)\ ;\ (Lift\ (s \mid x_1)_{+x2}\ ;\ A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; (Lift\ (s \mid x_2)_{+x1}\ ;\ A_2$
= [Parallel Distributivity C.2.37]
$Lift\ (s)\ ;\ (Lift\ (s \mid x_1)_{+x2}\ ;\ A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; Lift\ (s)\ ;\ (Lift\ (s \mid x_2)_{+x1}\ ;\ A_2$
= [Lift Composition (twice)C.2.27]
$Lift\ (s;\ (s \mid x_1)_{+x2}\ ;\ A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; Lift\ (s\ ;\ (s \mid x_2)_{+x1}\ ;\ A_2$
= [Lift Semi Idempotence C.2.38]
$Lift\ (s)\ ;\ A_1) \; [\![x_1 \mid cs \mid x_2]\!] \; Lift\ (s_2)\ ;\ A_2$
= [Lift Parallel Distributivity C.2.37]
$Lift\ (s)\ ;\ A_1 \; [\![x_1 \mid cs \mid x_2]\!] \; A_2$
= LHS □

Attached Rule 18.

Parallel End*:*

$(c \mid s \models (loc\ s_1 \bullet Skip) \; [\![x_1 \mid cs \mid x_2]\!] \; (loc\ s_2 \bullet Skip)) \xrightarrow{\tau} (c \mid (s_1 \mid x1) \wedge (s_2 \mid x_2) \models Skip)$

Proof

$(c \mid s \models (loc\ s_1 \bullet Skip) \; [\![x_1 \mid cs \mid x_2]\!] \; (loc\ s_2 \bullet Skip)) \xrightarrow{\tau} (c \mid (s_1 \mid x_1) \wedge (s_2 \mid x_2) \models Skip)$

= [Definition of Silent Transiton C.25]

$$\forall \text{ w . c} \Rightarrow \left(\begin{array}{l} \mathit{Lift}\,(s)\,;\ \mathit{loc}\,s_1 \bullet \mathit{Skip}\ [\![\, cs \,]\!]\ \mathit{loc}\,s_2 \bullet \mathit{Skip} \\ \sqsubseteq \mathit{Lift}\,(s)\,;\ (\mathit{loc}\,(s \mid x_1)_{+x2} \bullet \mathit{Skip})\ [\![\, x_1 \mid cs \mid x_2 \,]\!]\ (\mathit{loc}\,(s \mid x_2)_{+x1} \bullet \mathit{Skip}) \end{array} \right)$$

The proof from now on will consist on proving that the RHS of the refinement is equal to the LHS.

LHS

= Lift (s) ; (loc s_1 • Skip) [[x_1 | cs | x_2]] (loc s_2 • Skip)

= [Definition of Loc C.29]

Lift (s) ; (Lift (s_1) ; Skip) [[x_1 | cs | x_2]] (Lift (s_2) ; Skip)

= [Parallel Distributivity C.2.37]

(Lift (s) ; Lift(s_1) ; Skip) [[x_1 | cs | x_2]] (Lift (s) ; Lift(s_2) ; Skip)

= [Lift Left Unit C.2.29 twice]

(Lift (s_1) ; Skip) [[x_1 | cs | x_2]] (Lift (s_2) ; Skip)

= [Lift CSP4 C.2.26 twice]

Lift (s_1) [[x_1 | cs | x_2]] Lift (s_2)

= [Lift Merge C.2.39]

Lift (s_1 | x_1 ∧ s_2 | x_2) ; Skip

=

RHS □

Attached Rule 19.

Parallel Independent Left:

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3)\ \ ((l = \tau) \vee \mathit{chan}(l) \notin cs)}{\begin{array}{l}(c \mid s \models \mathit{loc}\ s_1 \bullet A_1\ [\![\, x_1 \mid CS \mid x_2 \,]\!]\ \mathit{loc}\ s_2 \bullet A_2) \\ \xrightarrow{l} (c \mid s \models \mathit{loc}\ s_3 \bullet A_3\ [\![\, x_1 \mid CS \mid x_2 \,]\!]\ \mathit{loc}\ s_2 \bullet A_2)\end{array}}$$

Recapitulating the assumptions given on rule 19, they are:

- (c_1 | s_1 ⊨ A_1) $\xrightarrow{l}$ (c_3 | s_3 ⊨ A_3)
- ((l = τ) ∨ chan(l) ∉ cs)

Beyond the previous two assumptions, we also provide the following assumptions (in order to apply Parallel Prefixed Loc 8 rule):

- Lift (s1) ; A1 ⊑ Lift (s3) ; l → A3 □ Lift (s1) ; A1 **[Assumption 3]**
- initials (Lift (s2) ; A2) ⊆ CS **[Assumption 4]**
- CS ∩ usedC (Lift (s3); A3) = ∅ **[Assumption 5]**
- wrtV (Lift (s3); A3) ∩ usedV (Lift (s2) ; A2) = ∅ **[Assumption 6]**

- Lift (s2) ; A2 is divergence free **[Assumption 7]**
- ok = ok' **[Assumption 8]**

From assumptions (c ⌈ s1 $\models$ A1) $\xrightarrow{l}$ (c3 ⌈ s3 $\models$ A3) and (l $\neq \tau$)

we apply Labelled Transition Definition and find

$\forall$ w . c $\wedge$ c3 $\Rightarrow$ Lift (s1) ; A1 $\sqsubseteq$ Lift (s3) ; A3

Then we consider both c and c3 being true and we find

$\forall$ w . true $\wedge$ true $\Rightarrow$ Lift (s1) ; A1 $\sqsubseteq$ Lift (s3) ; A3

then we apply Predicate Calculus twice and find

Lift (s1) ; A1 $\sqsubseteq$ Lift (s3) ; A3

For the case where c or c3 is false, we have a false assumption and then it makes the expression directly true ((false $\Rightarrow$ Rule) = true).

Proof
As there is an $\vee$ operator between factor l = τ and factor chan (l) $\notin$ CS, the proof must be done to both cases:

For l = τ:

(c ⌈ s $\models$ loc s_1 • A_1 $[\![x_1 \mid CS \mid x_2]\!]$ loc s_2 • A_2)
$\xrightarrow{l}$ (c3 ⌈ s $\models$ loc s_3 • A_3 $[\![x_1 \mid CS \mid x_2]\!]$ loc s_2 • A_2)
= [Definition of Silent Transition C.25 (as l = τ, the transition is silent)]
$\forall$ w . c $\wedge$ c3 $\Rightarrow$
Lift (s) ; (loc s1 • A1) $[\![$ x1 {| CS |} x2 $]\!]$ (loc s2 • A2)
$\sqsubseteq$ Lift (s) ; (loc s3 • A3) $[\![$ x1 ⌈ CS ⌈ x2 $]\!]$ (loc s2 • A2)
= [w is universally quantified, so we abstract it throughout the proof]
c $\wedge$ c3 $\Rightarrow$
Lift (s) ; (loc s1 • A1) $[\![$ x1 {| CS |} x2 $]\!]$ (loc s2 • A2)
$\sqsubseteq$ Lift (s) ; (loc s3 • A3) $[\![$ x1 ⌈ CS ⌈ x2 $]\!]$ (loc s2 • A2)

= [Par.Ind.Ref.9 (providing Assumpt.1 from the rule)+Ref.Eq.Sides 1+Ref.Mon.6] true

For chan (l) $\notin$ CS , l $\neq \tau$ and c and c3 are true
(when at least some of them is false the whole expression is true):

$(c \mid s \models \text{loc } s_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; \text{loc } s_2 \bullet A_2)$
$\xrightarrow{l} (c3 \mid s \models \text{loc } s_3 \bullet A_3 \; [\![x_1 \mid CS \mid x_2]\!] \; \text{loc } s_2 \bullet A_2)$
= [Definition of Labelled Transition C.26 (as l $\neq \tau$, the transition is labelled)]
$\forall w \,.\, c \wedge c3 \Rightarrow$
$$\left(\begin{array}{l} Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \sqsubseteq \\ \left(\begin{array}{l} Lift(s);\ l \rightarrow locs_3 \bullet A_3 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \Box\ Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \end{array} \right) \end{array} \right)$$
= [w is universally quantified, so we abstract it]
$c \wedge c3 \Rightarrow$
$$\left(\begin{array}{l} Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \sqsubseteq \\ \left(\begin{array}{l} Lift(s);\ l \rightarrow locs_3 \bullet A_3 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \Box\ Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \end{array} \right) \end{array} \right)$$
= [c and c3]
$true \Rightarrow$
$$\left(\begin{array}{l} Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \sqsubseteq \\ \left(\begin{array}{l} Lift(s);\ l \rightarrow locs_3 \bullet A_3 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \Box\ Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \end{array} \right) \end{array} \right)$$
= [Predicate Calculus]
$$\left(\begin{array}{l} Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \sqsubseteq \\ \left(\begin{array}{l} Lift(s);\ l \rightarrow locs_3 \bullet A_3 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \Box\ Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \end{array} \right) \end{array} \right)$$
= [External Choice Refinement Implication 15]
$$\left(\begin{array}{l} Lift(s);\ locs_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \\ \sqsubseteq \\ \left(Lift(s);\ l \rightarrow locs_3 \bullet A_3 \; [\![x_1 \mid CS \mid x_2]\!] \; locs_2 \bullet A_2 \right) \end{array} \right)$$
= [Parallel Prefixed Loc 8 (having assumptions 3 to 8) with Monotonicity of Refinement 6]

true
□

Attached Rule 20.

Parallel Independent Right:

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \; ((l = \tau) \vee chan(l) \notin cs)}{(c \mid s \models loc\ s_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; loc\ s_2 \bullet A_2) \xrightarrow{l} (c \mid s \models loc\ s_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; loc\ s_3 \bullet A_3)}$$

Recapitulating the previous assumptions:

- $(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3)$ **[Assumption 1]**
- $(l = \tau) \vee chan(l) \notin cs$ **[Assumption 2]**

Beyond the previous two assumptions, we also provide the following assumptions (in order to apply Parallel Prefixed Loc 8 rule):

- Lift (s2) ; A2 $\sqsubseteq$ Lift (s3) ; l $\rightarrow$ A3 $\square$ Lift (s2) ; A2 **[Assumption 3]**
- initials (Lift (s1) ; A1) $\subseteq$ CS **[Assumption 4]**
- CS $\cap$ usedC (Lift (s3); A3) = $\emptyset$ **[Assumption 5]**
- wrtV (Lift (s3); A3) $\cap$ usedV (Lift (s1) ; A1) = $\emptyset$ **[Assumption 6]**
- Lift (s1) ; A1 is divergence free **[Assumption 7]**
- ok = ok' **[Assumption 8]**

Proof

$(c \mid s \models loc\ s_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; loc\ s_2 \bullet A_2)$
$\xrightarrow{l} (c \mid s \models loc\ s_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; loc\ s_3 \bullet A_3)$
= [Compound Actions Commutative C.2.6]
$(c \mid s \models loc\ s_2 \bullet A_2 \; [\![x_2 \mid CS \mid x_1]\!] \; loc\ s_1 \bullet A_1)$
$\xrightarrow{l} (c \mid s \models loc\ s_1 \bullet A_1 \; [\![x_1 \mid CS \mid x_2]\!] \; loc\ s_3 \bullet A_3)$
= [Compound Actions Commutative C.2.6]
$(c \mid s \models loc\ s_2 \bullet A_2 \; [\![x_2 \mid CS \mid x_1]\!] \; loc\ s_1 \bullet A_1)$
$\xrightarrow{l} (c \mid s \models loc\ s_3 \bullet A_3 \; [\![x_3 \mid CS \mid x_1]\!] \; loc\ s_1 \bullet A_1)$
= [Parallel Independent Left 19 (having assumptions 2-12)]
true □

Attached Rule 21.

Parallel Synchronised:

$$\frac{\begin{array}{l} d \in cs\ c_1\ c_2\ c_3\ c_4\ (w_1 = w_2) \\ (^*,\diamond) \in (?, !), (!, ?), (!, !), (., .), (?, ?) \\ (c_1 \mid s_1 \models A_1) \xrightarrow{d^* w_1} (c_3 \mid s_3 \models A_3) \\ (c_2 \mid s_2 \models A_2) \xrightarrow{d \diamond w_2} (c_4 \mid s_4 \models A_4) \end{array}}{\begin{array}{l} (c_1 \wedge c_2 \mid s \models (c_1 \mid loc\ s_1 \bullet A_1) \ [\![x_1 \mid cs \mid x_2]\!] \ (c_2 \mid loc\ s_2 \bullet A_2)) \\ \xrightarrow{d|w_2} \\ \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \models \\ \left(\begin{array}{l} (c_3 \wedge (w_1 = w_2) \mid locs_3 \bullet A_3) \\ [\![x_1 \mid cs \mid x_2]\!] \\ (c_4 \wedge (w_1 = w_2) \mid locs_4 \bullet A_4) \end{array} \right) \end{array} \right) \end{array}}$$

Recapitulating the previous assumptions:

- d ∈ cs, c_1, c_2, c_3, c_4 **[Assumptions 1, 2, 3, 4 and 5]**
- $w_1 = w_2$ **[Assumption 6]**
- (*,◇) ∈ (?, !), (!, ?), (!, !), (., .), (?, ?) **[Assumption 7]**
- $(c_1 \mid s_1 \models A_1) \xrightarrow{d^* w_1} (c_3 \mid s_3 \models A_3)$ **[Assumption 8]**
- $(c_2 \mid s_2 \models A_2) \xrightarrow{d \diamond w_2} (c_4 \mid s_4 \models A_4)$ **[Assumption 9]**

We also provide the following assumption in order to allow the proof using Lift Shift 7:

- 3.ok = 3.ok' and 4.ok = 4.ok' **[Assumption 10]**
  (3 and 4 are labels for some branches of the parallelism, such that i is the label for branch Lift (s_i) ; d → A_i);

And now we derive other assumptions from **Assumption 8** and **Assumption 9**, called **compldw1** and **compldw2**:
[Assumption 8]
⇒ **[Labelled Transition Implication C.17]**
∀ w. $c_1 \wedge c_3$ ⇒ (Lift (s_1); A_1) ⊑ (Lift (s_3); $d^* w_1$ → A_3) **[Assumption compldw1]**
= w is unviversally quantified, so we abstract it]
$c_1 \wedge c_3$ ⇒ (Lift (s_1); A_1) ⊑ (Lift (s_3); $d^* w_1$ → A_3) **[Assumption compldw1]**
[Assumption 9]
⇒ **[Labelled Transition Implication C.17]**
$c_1 \wedge c_3$ ⇒ (Lift (s_1); A_1) ⊑ (Lift (s_3); $d^* w_1$ → A_3) **[Assumption compldw2]**
Now we will start proving.

Proof

$(c_1 \wedge c_2 \mid s \models (c_1 \mid \text{loc } s_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \text{loc } s_2 \bullet A_2))$
$\xrightarrow{d|w_2}$
$(c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \models ((\text{loc } s_3 \bullet A_3) \,[\![x_1 \mid cs \mid x_2]\!]\, (\text{loc } s_4 \bullet A_4)))$

= [Definition of Labelled Transition]

$$\forall\, w \,.\, c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow \left(\begin{array}{l} (\mathit{Lift}(s);\ (\mathit{locs}_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{locs}_2 \bullet A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l} (\mathit{Lift}(s);\ d \mid w_2 \rightarrow ((\mathit{locs}_3 \bullet A_3) \,[\![x_1 \mid cs \mid x_2]\!]\, (\mathit{locs}_4 \bullet A_4))) \\ \Box \\ (\mathit{Lift}(s);\ (\mathit{locs}_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{locs}_2 \bullet A_2)) \end{array} \right) \end{array} \right)$$

= [w is universally quantified, so we abstract it]

$$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow \left(\begin{array}{l} (\mathit{Lift}(s);\ (\mathit{locs}_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{locs}_2 \bullet A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l} (\mathit{Lift}(s);\ d \mid w_2 \rightarrow ((\mathit{locs}_3 \bullet A_3) \,[\![x_1 \mid cs \mid x_2]\!]\, (\mathit{locs}_4 \bullet A_4))) \\ \Box \\ (\mathit{Lift}(s);\ (\mathit{locs}_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{locs}_2 \bullet A_2)) \end{array} \right) \end{array} \right)$$

= [Communication Parallelism Distribution C.2.7 (having that $\{d\} \in$ CS, as it is Par. Synchronised)]

$$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow \left(\begin{array}{l} (\mathit{Lift}(s);\ (\mathit{locs}_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{locs}_2 \bullet A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l} (\mathit{Lift}(s);\ (d \mid w_2 \rightarrow (\mathit{locs}_3 \bullet A_3) \,[\![x_1 \mid cs \mid x_2]\!]\, d \mid w_2 \rightarrow (\mathit{locs}_4 \bullet A_4))) \\ \Box \\ (\mathit{Lift}(s);\ (\mathit{locs}_1 \bullet A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{locs}_2 \bullet A_2)) \end{array} \right) \end{array} \right)$$

= [Definition of Loc C.29 6x]

$$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow \left(\begin{array}{l} (\mathit{Lift}(s);\ (\mathit{Lift}(s_1);\ A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{Lift}(s_2);\ A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l} (\mathit{Lift}(s);\ (d \mid w_2 \rightarrow (\mathit{Lift}(s_3);\ A_3) \,[\![x_1 \mid cs \mid x_2]\!]\, d \mid w_2 \rightarrow (\mathit{Lift}(s_4);\ A_4))) \\ \Box \\ (\mathit{Lift}(s);\ (\mathit{Lift}(s_1);\ A_1) \,[\![x_1 \mid cs \mid x_2]\!]\, (c_2 \mid \mathit{Lift}(s_2);\ A_2)) \end{array} \right) \end{array} \right)$$

= [Lift Shift 7 (having assumption 10)]

$$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow$$

$$\left(\begin{array}{l}(Lift(s);\ (Lift(s_1);\ A_1)\ [\![x_1 \mid cs \mid x_2]\!]\ (c_2 \mid Lift(s_2);\ A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l}(Lift(s);\ ((Lift(s_3);\ d \mid w_2 \to A_3)\ [\![x_1 \mid cs \mid x_2]\!]\ (Lift(s_4);\ d \mid w_2 \to A_4))) \\ \Box \\ (Lift(s);\ (Lift(s_1);\ A_1)\ [\![x_1 \mid cs \mid x_2]\!]\ (c_2 \mid Lift(s_2);\ A_2))\end{array}\right)\end{array}\right)$$

= [If $w_1 \neq w_2$, the antecedent is false, thus the whole expression is true. If $w_1 = w_2$ is true, w_2 can be replaced by w_1]

$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow$

$$\left(\begin{array}{l}(Lift(s);\ (Lift(s_1);\ A_1)\ [\![x_1 \mid cs \mid x_2]\!]\ (c_2 \mid Lift(s_2);\ A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l}(Lift(s);\ ((Lift(s_3);\ d \mid w_1 \to A_3)\ [\![x_1 \mid cs \mid x_2]\!]\ (Lift(s_4);\ d \mid w_2 \to A_4))) \\ \Box \\ (Lift(s);\ (Lift(s_1);\ A_1)\ [\![x_1 \mid cs \mid x_2]\!]\ (c_2 \mid Lift(s_2);\ A_2))\end{array}\right)\end{array}\right)$$

= [External Choice Refinement Implication 15]

$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow$

$$\left(\begin{array}{l}(Lift(s);\ (Lift(s_1);\ A_1)\ [\![x_1 \mid cs \mid x_2]\!]\ (c_2 \mid Lift(s_2);\ A_2)) \\ \sqsubseteq \\ \left(\begin{array}{l}(Lift(s);\ ((Lift(s_3);\ d \mid w_1 \to A_3)\ [\![x_1 \mid cs \mid x_2]\!]\ (Lift(s_4);\ d \mid w_2 \to A_4)))\end{array}\right)\end{array}\right)$$

= [Assumptions complw1 and complw2 with Monotonicity of Refinement for Parallelism C.2.21]

$c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge w_1 = w_2 \Rightarrow$ true

= [Predicate Calculus]

true

□

C.3.9 Hiding

Attached Rule 22.

Hiding Internal:

$$\frac{(c1 \mid s1 \models A) \xrightarrow{l} (c2 \mid s2 \models B) \quad l \in S}{(c1 \mid s1 \models A \setminus S) \xrightarrow{\tau} (c2 \mid s2 \models B \setminus S)}$$

For proving the above rule, we will also assume that (A \ S) and (B \ S) are divergence-free (ok and ok'), and we will call this assumption as **[Assumption Div]**.

On the rule, as it has a labelled transition from program text A to program text B, then A = l → B **[We will call it Assumption 3]**.

[Assumption 1]
(c1 ∣ s1 ⊨ A) $\xrightarrow{l}$ (c2 ∣ s2 ⊨ B)
= [Definition of Labelled Transition C.26]
∀w . c1 ∧ c2 ⇒ Lift (s1) ; A ⊑ (Lift (s2) ; l → B) □ Lift (s1) ; A
= [w is universally quantified, so we abstract it]
c1 ∧ c2 ⇒ Lift (s1) ; A ⊑ (Lift (s2) ; l → B) □ Lift (s1) ; A
⇒ **[External Choice Refinement Implication 15]**
c1 ∧ c2 ⇒ Lift (s1) ; A ⊑ (Lift (s2) ; l → B)
= [For c1 and c2 being true, we have as follows (if either c1 or c2 is false, the antecedent is false, thus the whole expression is true)]
true ⇒ Lift (s1) ; A ⊑ (Lift (s2) ; l → B)
= [Predicate Calculus]
Lift (s1) ; A ⊑ (Lift (s2) ; l → B)
= [Assumption 3]
Lift (s1) ; l → B ⊑ (Lift (s2) ; l → B)
⇒ **[Hiding Monotonic C.2.9]**
(Lift (s1) ; l → B) \ S ⊑ (Lift (s2) ; l → B) \ S
= [Sequence Skip C.2.2]
(Lift (s1) ; l → Skip ; B) \ S ⊑ (Lift (s2) ; l → Skip ; B) \ S
= [Hidden Event Sequenced By Skip 16 (having ok and ok' from Assumption Div)]
(Lift (s1) ; Skip ; B) \ S ⊑ (Lift (s2) ; Skip ; B) \ S
= [Sequence Skip C.2.2]
(Lift (s1) ; B) \ S ⊑ (Lift (s2) ; B) \ S **[Assumption 4]**

Expression to be proved:

$\forall w$. c1 $\wedge$ c2 $\Rightarrow$ Lift (s1) ; A $\setminus$ S $\sqsubseteq$ Lift (s2) ; B $\setminus$ S

Proof

(c1 ∣ s1 $\models$ A $\setminus$ S) $\xrightarrow{\tau}$ (c2 ∣ s2 $\models$ B $\setminus$ S)

= [Definition of Silent Transition C.25]

$\forall w$. c1 $\wedge$ c2 $\Rightarrow$ Lift (s1) ; A $\setminus$ S $\sqsubseteq$ Lift (s2) ; B $\setminus$ S

= [w is universally quantified, so we abstract it]

c1 $\wedge$ c2 $\Rightarrow$ Lift (s1) ; A $\setminus$ S $\sqsubseteq$ Lift (s2) ; B $\setminus$ S

= [By Assumption 3]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) ; (l $\rightarrow$ B)) $\setminus$ S $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Hiding Sequence Distributive C.2.12]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) ; (l $\rightarrow$ B)) $\setminus$ S $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Skip A Equals A C.2.2]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) ; (l $\rightarrow$ Skip ; B)) $\setminus$ S $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Hiding Sequence Distributive C.2.12]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) ; ((l $\rightarrow$ Skip) $\setminus$ S ; B $\setminus$ S)) $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Hidden Event Sequenced by Skip 16]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) ; (Skip ; B $\setminus$ S)) $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Skip A Equals A C.2.2]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) ; (B $\setminus$ S)) $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Hiding Identity C.2.8 (as Lift (s1) is a sequence of assignments, it does not contain any channel)]

c1 $\wedge$ c2 $\Rightarrow$ (Lift (s1) $\setminus$ S ; (B $\setminus$ S)) $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Assume c1 $\wedge$ c2 from now on (when c1 $\wedge$ c2 is false, then the whole expression is true)]

(Lift (s1) $\setminus$ S ; (B $\setminus$ S)) $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Hiding Sequence Distributive C.2.12]

(Lift (s1) ; B) $\setminus$ S $\sqsubseteq$ (Lift (s2) ; B) $\setminus$ S

= [Assumption 4]

true

□

Attached Rule 23.

***Hiding Visible**:*

$$\frac{(c1 \mid s1 \models A) \xrightarrow{l} (c2 \mid s2 \models B) \qquad l \notin S}{(c1 \mid s1 \models A \setminus S) \xrightarrow{l} (c2 \mid s2 \models B \setminus S)}$$

Expression to be proved:

$\forall$w. c1$\wedge$c2 $\Rightarrow$ Lift (s1); A\S $\sqsubseteq$ (Lift (s2); c.w1 $\rightarrow$ B\S) $\Box$ Lift (s1); A\S

Proof

$\forall$w . c1 $\wedge$ c2 $\Rightarrow$

Lift (s1) ; A $\setminus$ S $\sqsubseteq$ (Lift (s2) ; c.w1 $\rightarrow$ B $\setminus$ S) $\Box$ Lift (s1) ; A $\setminus$ S

= [w is universally quantified, so we abstract it]

c1 $\wedge$ c2 $\Rightarrow$

Lift (s1) ; A $\setminus$ S $\sqsubseteq$ (Lift (s2) ; c.w1 $\rightarrow$ B $\setminus$ S) $\Box$ Lift (s1) ; A $\setminus$ S

= [No channels are used on Lift (s), so Lift (s) = Lift (s) $\setminus$ S]

c1 $\wedge$ c2 $\Rightarrow$

Lift (s1) $\setminus$ S ; A $\setminus$ S $\sqsubseteq$ (Lift (s2) $\setminus$ S ; c.w1 $\rightarrow$ B $\setminus$ S) $\Box$ Lift (s1) $\setminus$ S ; A $\setminus$ S

= [Hiding Sequence Distributive C.2.12]

c1 $\wedge$ c2 $\Rightarrow$

(Lift (s1) ; A) $\setminus$ S $\sqsubseteq$ ((Lift (s2) ; c.w1 $\rightarrow$ B) $\setminus$ S) $\Box$ (Lift (s1) ; A) $\setminus$ S

= [Hiding External Choice Distributive C.2.10]

c1 $\wedge$ c2 $\Rightarrow$

(Lift (s1) ; A) $\setminus$ S $\sqsubseteq$ ((Lift (s2) ; c.w1 $\rightarrow$ B) $\Box$ (Lift (s1) ; A)) $\setminus$ S

= [Assumption and Hiding Monotonic C.2.9]

c1 $\wedge$ c2 $\Rightarrow$ true

= [Predicate Calculus]

true $\Box$

Attached Rule 24.

***Hiding Skip**:*

$(c \mid s \models Skip \setminus S) \xrightarrow{\tau} (c \mid s \models Skip)$

Proof We start this proof by infering an assumption from function UsedC (C.12):

(UsedC (Skip) = { }) $\Rightarrow$ (New Assumption) (**S** $\cap$ **UsedC (Skip) =** { })

Now we will continue the proof:

(c | s $\models$ Skip $\setminus$ S) $\xrightarrow{\tau}$ (c | s $\models$ Skip)

= [From New Assumption and Law C.2.8]

$(c \mid s \models \text{Skip}) \xrightarrow{\tau} (c \mid s \models \text{Skip})$

= [Silent Transition between Equivalent nodes 2]

true □

C.3.10 Recursion

Attached Rule 25.

Recursion:

$$\frac{(c \mid s \models A) \xrightarrow{\tau} (c \mid s \models B)\ N = A}{(c \mid s \models N) \xrightarrow{\tau} (c \mid s \models B)}$$

Proof The proof for Recursion is straightforward. We only apply the assumption N = A and then the other assumption to prove:
(c | s $\models$ N) $\xrightarrow{\tau}$ (c | s $\models$ B)
= [Assumption N = A]
(c | s $\models$ A) $\xrightarrow{\tau}$ (c | s $\models$ B)
= [Assumption (c | s $\models$ A) $\xrightarrow{\tau}$ (c | s $\models$ B)]
true □

C.3.11 Call

Attached Rule 26.

Call:

$$\frac{CALL = Content\ (CALL)}{(c \mid s \models CALL) \xrightarrow{\tau} (c \mid s \models Content\ (CALL))}$$

Proof (c | s $\models$ CALL) $\xrightarrow{\tau}$ (c | s $\models$ Content (CALL))
= [Assumption CALL = Content (CALL)]
(c | s $\models$ Content (CALL)) $\xrightarrow{\tau}$ (c | s $\models$ Content (CALL))
= [Silent Transition Between Equivalent Nodes 2]
true □

C.3.12 Iterated Actions

Attached Rule 27.

Iterated Actions:
(c | s $\models$ OP Decl • A) $\xrightarrow{\tau}$ (c | s $\models$ IteratedExpansion (A, Decl, ITOPFLAG))
IteratedExpansion is defined on C.3. ITOPFLAG, as indicated on C.3, is a flag that indicates the operator that is being iterated.

Proof. (c | s $\models$ OP Decl $\bullet$ A) $\xrightarrow{\tau}$ (c | s $\models$ IteratedExpansion (A, Decl, ITOPFLAG))
= [Definition of Iterated Operator C.3]
(c | s $\models$ IteratedExpansion (A, Decl, ITOPFLAG)) $\xrightarrow{\tau}$ (c | s $\models$ IteratedExpansion (A, Decl, ITOPFLAG))
= [Silent Transition Equal Sides 2]
true □

C.3.13 If-Guarded Command

Attached Rule 28.

***If-Guarded Command**:*
Be

$IGC \mathrel{\widehat{=}} \textbf{if } (pred_1) \rightarrow A_1 \; [] \; (pred_2) \rightarrow A_2 \; [] \; ... \; [] \; (pred_n) \rightarrow A_n \textbf{ fi}$, *then*

$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_1 \mid s \models A_1) \; \boldsymbol{T}_1$
$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_2 \mid s \models A_2) \; \boldsymbol{T}_2$
...
$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_n \mid s \models A_n) \; \boldsymbol{T}_n$

$(c \mid s \models IGC)$
$\xrightarrow{\tau}$
$(c \wedge (\neg \; pred_1) \wedge (\neg \; pred_2) \wedge ... \wedge (\neg \; pred_{n-1}) \wedge (\neg \; pred_n) \mid s \models Chaos) \; \boldsymbol{T}_{Chaos}$

Proof

Proof of $\boldsymbol{T}_1$

Among rules T_i, we will prove only law T_1. The other laws can be proved using similar reasoning.

$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge pred_1 \mid s \models A_1)$
= [Definition of Silent Transition C.25]
$\forall$ w . c $\wedge$ c $\wedge$ $pred_1$ $\Rightarrow$ (Lift (s) ; IGC $\sqsubseteq$ Lift (s) ; A_1)
= [w is universally quantified, so we abstract it]
c $\wedge$ c $\wedge$ $pred_1$ $\Rightarrow$ (Lift (s) ; IGC $\sqsubseteq$ Lift (s) ; A_1)
= [Monotonicity of Refinement 6: if Lift (s) $\sqsubseteq$ Lift (s) and IGC $\sqsubseteq$ A_1 (by C.12 and (assms) $pred_1$ ($pred_1$ is true because it appears on the left side of the implication))]
c $\wedge$ c $\wedge$ $pred_1$ $\Rightarrow$ true
= [Predicate Calculus]
true

Now we will prove T_{Chaos} rule.

Proof of <T_{Chaos}> rule:

$(c \mid s \models IGC) \xrightarrow{\tau} (c \wedge (\neg\ pred_1) \wedge ... \wedge (\neg\ pred_n) \mid s \models Chaos)$

=

$\forall\ w\ .\ (c \wedge c \wedge (\neg\ pred_1) \wedge ... \wedge (\neg\ pred_n)) \Rightarrow (Lift\ (s)\ ;\ IGC \sqsubseteq Lift\ (s)\ ;\ Chaos)$

= [w is universally quantified, so we abstract it]

$(c \wedge c \wedge (\neg\ pred_1) \wedge ... \wedge (\neg\ pred_n)) \Rightarrow (Lift\ (s)\ ;\ IGC \sqsubseteq Lift\ (s)\ ;\ Chaos)$

= [c is true]

$((\neg\ pred_1) \wedge ... \wedge (\neg\ pred_n)) \Rightarrow (Lift\ (s)\ ;\ IGC \sqsubseteq Lift\ (s)\ ;\ Chaos)$

= [$(\neg$ pred$_1$) $\wedge$ ... $\wedge$ ($\neg$ pred$_n$) $\Rightarrow$ (IGC = Chaos)]

$((\neg\ pred_1) \wedge ... \wedge (\neg\ pred_n)) \Rightarrow (Lift\ (s)\ ;\ Chaos \sqsubseteq Lift\ (s)\ ;\ Chaos)$

= [E $\sqsubseteq$ E]

$((\neg\ pred_1) \wedge ... \wedge (\neg\ pred_n)) \Rightarrow true$

=

true

□

C.3.14 Z Schema

Attached Rule 29.

***Z Schema**:*

$$\frac{c \wedge (s\ ;\ \textbf{\textit{pre}}\ Op)}{(c \mid s \models Op) \xrightarrow{\tau} (c \wedge (s\ ;\ Op\ [w0/v'\,]) \mid s;\ v := w0 \models Skip)}$$

The **pre** operator is defined on C.2.

Proof

$\forall\ w\ .\ c \wedge c \wedge (s\ ;\ Op\ [w0/v'\,]) \Rightarrow ((Lift\ (s)\ ;\ Op) \sqsubseteq (Lift\ (s;\ v := w0)\ ;\ Skip))$

The proof is divided in two steps: we firstly prove the expression for Op = true, and then prove it for Op = false

For Op = true:

$\forall\ w\ .\ c \wedge c \wedge (s\ ;\ true\ [w0/v'\,]) \Rightarrow ((Lift\ (s)\ ;\ true) \sqsubseteq (Lift\ (s;\ v := w0)\ ;\ Skip))$

= [w is universally quantified, so we abstract it]

$c \wedge c \wedge (s\ ;\ true\ [w0/v'\,]) \Rightarrow ((Lift\ (s)\ ;\ true) \sqsubseteq (Lift\ (s;\ v := w0)\ ;\ Skip))$

= [Lift Composition C.2.27]

$c \wedge c \wedge (s\ ;\ true\ [w0/v'\,]) \Rightarrow ((Lift\ (s)\ ;\ true) \sqsubseteq (Lift\ (s);\ Lift\ (v := w0)\ ;\ Skip))$

= [By lemma C.11, (true ⊑ Lift (v := w0) ; Skip). By Ref.Eq.Sides 1, (Lift (s) ⊑ Lift (s)). By both C.11 and 1 and by Monotonicity of refinement, the refinement expression on the right side of the implication is true]

c ∧ c ∧ (s ; true [w0/v']) ⇒ true

= [Predicate Calculus]

true

For Op = false:

∀ w . c ∧ c ∧ (s ; false [w0/v']) ⇒ ((Lift (s) ; false) ⊑ (Lift (s; v := w0) ; Skip))

= [w is universally quantified, so we abstract it]

c ∧ c ∧ (s ; false [w0/v']) ⇒ ((Lift (s) ; false) ⊑ (Lift (s; v := w0) ; Skip))

= [false [w/x] = false]

c ∧ c ∧ (s ; false) ⇒ ((Lift (s) ; false) ⊑ (Lift (s; v := w0) ; Skip))

= [S Sequence False 5]

c ∧ c ∧ false ⇒ ((Lift (s) ; false) ⊑ (Lift (s; v := w0) ; Skip))

= [Predicate Calculus]

false ⇒ ((Lift (s) ; false) ⊑ (Lift (s; v := w0) ; Skip))

= [Predicate Calculus]

true

□

C.3.15 Specification Statements

Attached Rule 30.

Specification Statements (Rule 1):

$(c \mid s \models V : [Pre, Post]) \xrightarrow{\tau} (c \wedge Pre \mid s \models V\ [:]\ [Pre, Post])$

The Extra Statement ([:]) operator C.31 has the role of representing the Specification Statement in an intermediate state where its pre-condition is holded but its post-condition was not processed yet. The role of the Extra-statement operator is the same as the **let** C.28 operator on the rules of Variable Block.

Proof

(c ∣ s ⊨ V : [Pre, Post]) $\xrightarrow{\tau}$ (c ∧ Pre ∣ s ⊨ V [:] [Pre, Post])

= [Extra statement (definition C.31)]

(c ∣ s ⊨ V : [Pre, Post]) $\xrightarrow{\tau}$ (c ∧ Pre ∣ s ⊨ V : [Pre, Post])

= [Silent Transition Equal Sides (lemma 2)]

true □

Attached Rule 31.

***Specification Statements (Rule 2)**:*
(c ∣ s ⊨ V : [Pre, Post]) $\xrightarrow{\tau}$ (c ∧ (¬ Pre) ∣ s ⊨ Skip)

Proof
(c ∣ s ⊨ V : [Pre, Post]) $\xrightarrow{\tau}$ (c ∧ (¬ Pre) ∣ s ⊨ Skip)
= [Skip V G True C.2.1]
∀ w . c ∧ c ∧ (¬ Pre) ⇒ Lift (s) ; V : [Pre, Post] ⊑ Lift (s) ; V : [true, true]

From now on, the proof will be made for all 4 different situations of Pre and Post:

Pre = Post = true:

∀ w . c ∧ c ∧ (¬ true) ⇒ Lift (s) ; V : [true, true] ⊑ Lift (s) ; V : [true, true]
= [w is universally quantified, so we abstract it]
c ∧ c ∧ (¬ true) ⇒ Lift (s) ; V : [true, true] ⊑ Lift (s) ; V : [true, true]
= [Predicate Calculus]
false ⇒ true
= [Predicate Calculus]
true

Pre = true, Post = false:

∀ w . c ∧ c ∧ (¬ true) ⇒ Lift (s) ; V : [true, false] ⊑ Lift (s) ; V : [true, true]
= [w is universally quantified, so we abstract it]
c ∧ c ∧ (¬ true) ⇒ (Lift (s) ; V : [true, false] ⊑ Lift (s) ; V : [true, true])
= false ⇒ (Lift (s) ; V : [true, false] ⊑ Lift (s) ; V : [true, true])
= true

Pre = false, Post = true:

(c ∣ s ⊨ V : [false, true]) $\xrightarrow{\tau}$ (c ∧ (¬ false) ∣ s ⊨ Skip)

= [Definition of Silent Transition C.25]

$\forall$ w . c $\wedge$ c $\wedge$ ($\neg$ false) $\Rightarrow$ Lift (s) ; V : [false, true] $\sqsubseteq$ Lift (s) ; V : [true, true]

= [w is universally quantified, so we abstract it]

c $\wedge$ c $\wedge$ ($\neg$ false) $\Rightarrow$ Lift (s) ; V : [false, true] $\sqsubseteq$ Lift (s) ; V : [true, true]

= [Assume c = true]

Lift (s) ; V : [false, true] $\sqsubseteq$ Lift (s) ; V : [true, true]

= [As Lift (s) is refined by itself, so, from Monotonicity of Refinement, the proof from now on will consist only on proving that V : [false, true] $\sqsubseteq$ V : [true, true]]

V : [false, true] $\sqsubseteq$ V : [true, true]

= [Specification Statement Denotational Definition B.6]

R (false $\vdash$ true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u) $\sqsubseteq$ R(true $\vdash$ true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

= [Predicate Calculus]

R ((false $\wedge$ ok)

$\Rightarrow$

((true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u) $\wedge$ ok') $\sqsubseteq$ R((true $\wedge$ ok) $\Rightarrow$ (true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u $\wedge$ ok'))

= [Predicate Calculus]

R (true)

$\sqsubseteq$ R((true $\wedge$ ok) $\Rightarrow$ (true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u $\wedge$ ok'))

= [Predicate Calculus]

R (true)

$\sqsubseteq$ R(($\neg$true $\vee$ $\neg$ok) $\vee$ (true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u $\wedge$ ok'))

= [Predicate Calculus]

R (true)

$\sqsubseteq$ R(($\neg$ok) $\vee$ ($\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u $\wedge$ ok'))

= [Be x = ($\neg$ok) $\vee$ ($\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u $\wedge$ ok')]

R (true) $\sqsubseteq$ R(x)

= [Healthy True Refinement C.6]

true

Pre = Post = false:

(c | s $\models$ V : [Pre,Post])$\xrightarrow{\tau}$(c $\wedge$ ($\neg$Pre) | s $\models$ Skip)

= [Definition of Silent Transition C.25]

$\forall$ w . c $\wedge$ c $\wedge$ ($\neg$ false) $\Rightarrow$ Lift (s) ; V : [false, false] $\sqsubseteq$ Lift (s) ; V : [true, true]

= [w is universally quantified, so we abstract it]

c $\wedge$ c $\wedge$ ($\neg$ false) $\Rightarrow$ Lift (s) ; V : [false, false] $\sqsubseteq$ Lift (s) ; V : [true, true]

= [Predicate Calculus]

Lift (s) ; V : [false, false] $\sqsubseteq$ Lift (s) ; V : [true, true]

= [Monotonicity of Refinement 6]

V : [false, false] $\sqsubseteq$ V : [true, true]

= [Predicate Calculus]

R (false $\vdash$ false $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

$\sqsubseteq$ R(true $\vdash$ true $\wedge$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

= [Predicate Calculus]

R (false $\vdash$ false) $\sqsubseteq$ R(true $\vdash$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

= [Predicate Calculus]

R (false $\wedge$ ok $\Rightarrow$ false $\wedge$ ok') $\sqsubseteq$ R(true $\vdash$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

= [Predicate Calculus]

R (false $\Rightarrow$ false) $\sqsubseteq$ R(true $\vdash$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

= [Predicate Calculus]

R (true) $\sqsubseteq$ R(true $\vdash$ $\neg$ wait' $\wedge$ tr' = tr $\wedge$ u' = u)

= [Healthy True Refinement C.6]

true

□

Attached Rule 32.

Specification Statements (Rule 3)*:*

(c $\wedge$ Pre | s $\models$ V [:] [Pre, Post])

$\xrightarrow{\tau}$

(c $\wedge$ Pre $\wedge$ Post $\wedge$ (α(P) - V)' = (α(P) - V)) | s $\models$ Skip)

Proof

(c $\wedge$ Pre | s $\models$ V [:] [Pre, Post])

$\xrightarrow{\tau}$

(c $\wedge$ Pre $\wedge$ (Post $\wedge$ (α(P) - V)' = (α(P) - V)) | s $\models$ Skip)

= [Definition of Silent Transition C.25]

$\forall$ w . (c $\wedge$ Pre $\wedge$ c $\wedge$ Pre $\wedge$ Post $\wedge$ (α(P) - V)' = (α(P) - V))

$\Rightarrow$ Lift (s) ; V [:] [Pre, Post] $\sqsubseteq$ Lift (s) ; Skip

We will prove the above expression checking the following complementary situations:

(Pre = false $\vee$ Post = false) and (Pre = true $\wedge$ Post = true)

Pre = false ∨ Post = false

In this case, the antecedent of the implication is false:

∀ w . (c ∧ false ∧ c ∧ Pre ∧ Post ∧ (α(P) - V)' = (α(P) - V))
⇒ Lift (s) ; V [:] [Pre, Post] ⊑ Lift (s) ; Skip
= [w is universally quantified, so we abstract it]
(c ∧ false ∧ c ∧ Pre ∧ Post ∧ (α(P) - V)' = (α(P) - V))
⇒ Lift (s) ; V [:] [Pre, Post] ⊑ Lift (s) ; Skip
= [Predicate Calculus]
(false ⇒ Lift (s) ; V [:] [Pre, Post] ⊑ Lift (s) ; Skip
= [Predicate Calculus]
true

Pre = true ∧ Post = true
∀ w . (c ∧ true ∧ c ∧ true ∧ true ∧ (α(P) - V)' = (α(P) - V))
⇒ Lift (s) ; V [:] [true, true] ⊑ Lift (s) ; Skip
= [w is universally quantified, so we abstract it]
(c ∧ true ∧ c ∧ true ∧ true ∧ (α(P) - V)' = (α(P) - V))
⇒ Lift (s) ; V [:] [true, true] ⊑ Lift (s) ; Skip
= [Predicate Calculus]
(c ∧ true ∧ c ∧ true ∧ true ∧ (α(P) - V)' = (α(P) - V))
⇒ Lift (s) ; V [:] [true, true] ⊑ Lift (s) ; V : [true, true]
= [Predicate Calculus]
(c ∧ true ∧ c ∧ true ∧ true ∧ (α(P) - V)' = (α(P) - V))
⇒ Lift (s) ; V : [true, true] ⊑ Lift (s) ; V : [true, true]
= [Refinement Equal Sides 1]
(c ∧ true ∧ c ∧ true ∧ true ∧ (α(P) - V)' = (α(P) - V))
⇒ true
= [Predicate Calculus]
= true
□

C.3.16 Basic Process

Attached Rule 33.

Basic Process Begin:

$$\frac{(c1 \mid s1 \models A1) \xrightarrow{lp} (c2 \mid s2 \models A2)}{(c1 \mid s1 \mid \textbf{\textit{begin state}}\ [Vars\text{-}decl \mid inv] \bullet A1\ \textbf{\textit{end}}) \xrightarrow{lp} (c2 \mid s2 \mid \textbf{\textit{begin state}}\ [Vars\text{-}decl \mid inv] \bullet A2\ \textbf{\textit{end}})}$$

Assumption:

$(c1 \mid s1 \models A1) \xrightarrow{lp} (c2 \mid s2 \models A2)$

= [Definition of Labelled Transition C.26]

$$\forall w \,.\, c1 \wedge c2 \Rightarrow \left(\begin{array}{l} Lift\,(s1)\,;\ A1 \\ \sqsubseteq Lift\,(s2)\,;\ lp \rightarrow A1 \ \Box\ Lift\,(s1)\,;\ A1 \end{array} \right)$$

⇒ [Labelled Transition Implication C.17]

$(\text{Lift } (s1); A1) \sqsubseteq (\text{Lift } (s3); \text{lp} \rightarrow A3)$ **[Assumption 2]**

Proof

$(c1 \mid s1 \mid \textbf{begin state}\ [Vars\text{-}decl \mid inv] \bullet A1\ \textbf{end}) \xrightarrow{lp} (c2 \mid s2 \mid \textbf{begin state}\ [Vars\text{-}decl \mid inv] \bullet A2\ \textbf{end})$

= [Definition of Labelled Transition C.26]

$$\forall w.c1 \wedge c2 \Rightarrow$$

$$Lift(s1);\ (\textbf{begin state}[Vars-decl \mid inv] \bullet A1 \textbf{end})$$

$$\sqsubseteq$$

$$\left(\begin{array}{l} Lift(s2);\ lp \rightarrow \textbf{begin state}[Vars-decl \mid inv] \bullet A1\ \textbf{end} \\ \Box\ Lift(s1);\ \textbf{begin state}[Vars-decl \mid inv] \bullet A1\ \textbf{end} \end{array} \right)$$

= [Basic Process Denotational Meaning C.2.15]

$$\forall w.c1 \wedge c2 \Rightarrow$$

$$Lift(s1);\ Vars-Decl \bullet A1$$

$$\sqsubseteq$$

$$\left(\begin{array}{l} Lift(s2);\ lp \rightarrow Vars-Decl \bullet A1 \\ \Box\ Lift(s1);\ Vars-Decl \bullet A1 \end{array} \right)$$

= [w is universally quantified, so we abstract it]

$$c1 \wedge c2 \Rightarrow$$

$$Lift(s1);\ Vars-Decl \bullet A1$$

$$\sqsubseteq$$

$$\left(\begin{array}{l} Lift(s2);\ lp \rightarrow Vars - Decl \bullet A1 \\ \Box\ Lift(s1);\ Vars - Decl \bullet A1 \end{array}\right)$$

= [External Choice Idempotence C.2.24]

$c1 \wedge c2 \Rightarrow$

$(Lift(s1);\ Vars - Decl \bullet A1) \Box (Lift(s1);\ Vars - Decl \bullet A1)$

$\sqsubseteq$

$$\left(\begin{array}{l} Lift(s2);\ lp \rightarrow Vars - Decl \bullet A1 \\ \Box\ Lift(s1);\ Vars - Decl \bullet A1 \end{array}\right)$$

= [Assumption 2 + External Choice Monotonic 14]

$c1 \wedge c2 \Rightarrow true$

= [Predicate Calculus]

true □

Attached Rule 34.

***Basic Process Reduction**:*

Based on C.2.16 and C.2.17 from the refinement calculus of Circus:

$$\frac{c \qquad \alpha(STA) \cap \alpha(STB) = \emptyset}{(c \models (\textbf{begin state } STA\ PARS\text{-}A \bullet A \textbf{ end})\ OP\ (\textbf{begin state } STB\ PARS\text{-}B \bullet B \textbf{ end}))}$$

$\rightsquigarrow$

$(c \mid s \models \textbf{begin state } (STA \wedge STB)\ (PARS\text{-}A \wedge_{\Xi} STB)\ (PARS\text{-}B \wedge_{\Xi} STA) \bullet A\ OP\ B \textbf{ end})$

Where $OP \in \{ \sqcap, \Box, ;, |||, [\![CS]\!], [\![\alpha(A) \mid CS \mid \alpha(B)]\!] \}$

As Basic Process Reduction is an abstract rule applied for a set of binary operators that appear as processes and as actions (Internal Choice, External Choice, Parallelism, Sequence, Interleaving), the proof involves the application of different refinement laws, depending on the operator. But the sequence of proof sub-goals is the same. As the proof is based on the application of rules C.2.16 and C.2.17. Be

B = $(\textbf{begin state } STA\ PARS - A \bullet A \textbf{ end})\ OP\ (\textbf{begin state } STB\ PARS - B \bullet B \textbf{ end})$, and

P = $\textbf{begin state } (STA \wedge STB)\ (PARS - A \wedge_{\Xi} STB)\ (PARS - B \wedge_{\Xi} STA) \bullet A\ OP\ B \textbf{ end})$

Proof

$$\left(\begin{array}{l}(c \models (\textbf{begin state}\ STA\ PARS - A \bullet A\ \textbf{end})\ OP\ (\textbf{begin state}\ STB\ PARS - B \bullet B\ \textbf{end})) \\ \rightsquigarrow \\ (c \mid s \models \textbf{begin state}\ (STA \wedge STB)\ (PARS - A \wedge_{\Xi} STB)\ (PARS - B \wedge_{\Xi} STA) \bullet A\ OP\ B\ \textbf{end})\end{array}\right)$$

= [Sigma Equal Basic Process (having, from def.C.2.16 and C.2.17, that B=P]

true

□

C.3.17 Compound Process

Attached Rule 35.

Compound Process Left:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3)}{(c_1 \models P_1 \; OP \; P_2) \rightsquigarrow (c_2 \models P_3 \; OP \; P_2)}$$

Where $OP \in \{ \sqcap, \square, ; , |||, [\![CS]\!] \}$

The proof will be divided on the following steps:

- Derivation of the assumption on other minor assumptions;
- Creation of a Lemma to synthesize the proof (the lemmas are referenced on the tactics and have their proofs made on appendix C.2.2);
- Division of the rule on a Sub-rule for each operator OP (OP $\in \{ \sqcap, \square, ; , |||, [\![CS]\!] \}$), and proof for each Sub-rule;

To reduce verbose, we create the abbreviation $gA(s)$ such that
$gA(s) = getAssignments(s)$

Assumptions C.19. .
[Assumption 1]
$(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3)$ =
[Definition of Syntactic Transition having w universally quantified C.27]
$c_1 \wedge c_2 \Rightarrow$
$((Lift\ (gA\ (P_1))\ ;\ P_1 \sqsubseteq Lift\ (gA\ (P_3))\ ;\ P_3)) \wedge Lift\ (gA\ (P_1)) = Lift\ (gA\ (P_3))$

$((Lift\ (gA\ (P_1))\ ;\ P_1 \sqsubseteq Lift\ (gA\ (P_3))\ ;\ P_3)) \wedge Lift\ (gA\ (P_1)) = Lift\ (gA\ (P_3))$
$\Rightarrow$
[Assumption 2] $Lift\ (gA\ (P_1)) = Lift\ (gA\ (P_3))$
$((Lift\ (gA\ (P_1))\ ;\ P_1 \sqsubseteq Lift\ (gA\ (P_3))\ ;\ P_3)) \wedge Lift\ (gA\ (P_1)) = Lift\ (gA\ (P_3)) \Rightarrow$
[Assumption 3] $Lift\ (gA\ (P_1))\ ;\ P_1 \sqsubseteq Lift\ (gA\ (P_3))\ ;\ P_3)$
As ***[Assumption 2]*** *and* ***[Assumption 3]***
then
[Assumption 4] $Lift\ (gA\ (P_3))\ ;\ P_1 \sqsubseteq Lift\ (gA\ (P_3))\ ;\ P_3$
As ***[Assumption 2]*** *and* ***[Assumption 4]***
[Assumption 5] $P_1 \sqsubseteq P_3$

Sub-Rule 1.

Internal Choice Compound Process Left:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = \sqcap}{(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)}$$

Proof

$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)$

= [Lemma Compound Process C.9]

$c_1 \wedge c_2 \Rightarrow$

$(Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2))\ ;\ P_1\ OP\ P_2 \sqsubseteq (Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2))\ ;\ P_3\ OP\ P_2))$

= [Be L = (Lift (gA (P_1)) $\wedge$ Lift (gA (P_2))]

$c_1 \wedge c_2 \Rightarrow$

$L\ ;\ P_1\ OP\ P_2 \sqsubseteq L\ ;\ P_3\ OP\ P_2$

= [Assumption OP = $\sqcap$]

$c_1 \wedge c_2 \Rightarrow$

$L\ ;\ P_1 \sqcap P_2 \sqsubseteq (L\ ;\ P_3 \sqcap P_2)$

= [$P_1 \sqsubseteq P_3$ (Assumption 5) $\wedge$ $P_2 \sqsubseteq P_2$, so $P_1 \sqcap P_2 \sqsubseteq P_3 \sqcap P_2$ (C.2.20). $L \sqsubseteq L \wedge P_1 \sqcap P_2 \sqsubseteq P_3 \sqcap P_2$, then (Monotonicity of Refinement 6) $L\ ;\ P_1 \sqcap P_2 \sqsubseteq L\ ;\ P_3 \sqcap P_2$]

$c_1 \wedge c_2 \Rightarrow$ true

= [Predicate Calculus: (P $\Rightarrow$ true) = true]

true

= true □

Sub-Rule 2.

External Choice Compound Process Left:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = \Box}{(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)}$$

Proof

$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)$

= [Lemma Compound Process C.9]

$c_1 \wedge c_2 \Rightarrow$

$(Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2))\ ;\ P_1\ OP\ P_2 \sqsubseteq (Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2))\ ;\ P_3\ OP\ P_2))$

= [Be L = (Lift (gA (P_1)) $\wedge$ Lift (gA (P_2))]

$c_1 \wedge c_2 \Rightarrow$

$L\ ;\ P_1\ OP\ P_2 \sqsubseteq L\ ;\ P_3\ OP\ P_2$

= [Assumption OP = □]

$c_1 \wedge c_2 \Rightarrow$

$L ; P_1 \Box P_2 \sqsubseteq (L ; P_3 \Box P_2)$
= [$P_1 \sqsubseteq P_3$ (Assumption 5) $\wedge$ $P_2 \sqsubseteq P_2$, so $P_1 \Box P_2 \sqsubseteq P_3 \Box P_2$ (C.2.22). $L \sqsubseteq L \wedge P_1 \Box P_2 \sqsubseteq P_3 \Box P_2$, then (Monotonicity of Refinement 6) $L ; P_1 \Box P_2 \sqsubseteq L ; P_3 \Box P_2$]
$c_1 \wedge c_2 \Rightarrow$ true
= ($P \Rightarrow$ true) = true
true
= true □

Sub-Rule 3.

Sequence Compound Process Left:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = ;}{(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)}$$

Proof
$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)$
= [Lemma Compound Process C.9]
$c_1 \wedge c_2 \Rightarrow$
$(Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2)) ; P_1\ OP\ P_2 \sqsubseteq (Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2)) ; P_3\ OP\ P_2))$
= [Be $L = (Lift\ (gA\ (P_1)) \wedge Lift\ (gA\ (P_2))$]
$c_1 \wedge c_2 \Rightarrow$
$L ; P_1\ OP\ P_2 \sqsubseteq L ; P_3\ OP\ P_2$
= [Assumption OP = ;]
$c_1 \wedge c_2 \Rightarrow$
$L ; P_1 ; P_2 \sqsubseteq (L ; P_3 ; P_2)$
= [$P_1 \sqsubseteq P_3$ (Assumption 5) $\wedge$ $P_2 \sqsubseteq P_2$, so $P_1 ; P_2 \sqsubseteq P_3 ; P_2$ (6)]. $L \sqsubseteq L \wedge P_1 ; P_2 \sqsubseteq P_3 ; P_2$, then (Monotonicity of Refinement 6) $L ; P_1 ; P_2 \sqsubseteq L ; P_3 ; P_2$]
$c_1 \wedge c_2 \Rightarrow$ true
= ($P \Rightarrow$ true) = true
true
= true □

Sub-Rule 4.

Parallelism Compound Process Left:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = [\![CS]\!]}{(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)}$$

Proof
$(c_1 \models P_1\ OP\ P_2) \rightsquigarrow (c_2 \models P_3\ OP\ P_2)$

= [Lemma Compound Process C.9]

$c_1 \wedge c_2 \Rightarrow$

$(\text{Lift } (\text{gA } (P_1)) \wedge \text{Lift } (\text{gA } (P_2)) \; ; \; P_1 \text{ OP } P_2 \sqsubseteq (\text{Lift } (\text{gA } (P_1)) \wedge \text{Lift } (\text{gA } (P_2)) \; ; \; P_3 \text{ OP } P_2))$

= [Be L = (Lift (gA (P_1)) $\wedge$ Lift (gA (P_2))]

$c_1 \wedge c_2 \Rightarrow$

$L \; ; \; P_1 \text{ OP } P_2 \sqsubseteq L \; ; \; P_3 \text{ OP } P_2$

= [Assumption OP = $[\![$ CS $]\!]$]

$c_1 \wedge c_2 \Rightarrow$

$L \; ; \; P_1 \; [\![\text{CS}]\!] \; P_2 \sqsubseteq (L \; ; \; P_3 \; [\![\text{CS}]\!] \; P_2)$

= [$P_1 \sqsubseteq P_3$ (Assumption 5) $\wedge$ $P_2 \sqsubseteq P_2$, so P_1 $[\![$ CS $]\!]$ $P_2 \sqsubseteq P_3$ $[\![$ CS $]\!]$ P_2 (C.2.21). L $\sqsubseteq$ L $\wedge$ P_1 $[\![$ CS $]\!]$ $P_2 \sqsubseteq P_3$ $[\![$ CS $]\!]$ P_2, then (Monotonicity of Refinement 6) L ; P_1 $[\![$ CS $]\!]$ P_2 $\sqsubseteq$ L ; P_3 $[\![$ CS $]\!]$ P_2]

$c_1 \wedge c_2 \Rightarrow \text{true}$

= ($P \Rightarrow$ true) = true

true

= true □

Sub-Rule 5.

Interleave Compound Process Left:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = |||}{(c_1 \models P_1 \; OP \; P_2) \rightsquigarrow (c_2 \models P_3 \; OP \; P_2)}$$

Proof

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = |||}{(c_1 \models P_1 \text{ OP } P_2) \rightsquigarrow (c_2 \models P_3 \text{ OP } P_2)}$$

= [Interleaving is equivalent to Parallelism with an empty channel set]

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = [\![\{ \}]\!]}{(c_1 \models P_1 \text{ OP } P_2) \rightsquigarrow (c_2 \models P_3 \text{ OP } P_2)}$$

= Parallelism Compound Process Left [4]

true □

Attached Rule 36.

Compound Process Right (for OP $\in \{ \Box, \sqcap, [\![CS]\!], ||| \}$***):***

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP \in \{ \Box, \sqcap, [\![CS]\!], ||| \}}{(c_1 \models P_1 \; OP \; P_2) \rightsquigarrow (c_2 \models P_1 \; OP \; P_3)}$$

As, except for Sequence, all compound operators are commutative, the proof for attached rule 36 will be done re-using attached rule 35 as a theorem. The proof for

Sequence will be done on sub-rule 6.

Proof

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP \in \{ \Box, \sqcap, [\![CS]\!], ||| \}}{(c_1 \models P_1 \ OP \ P_2) \rightsquigarrow (c_2 \models P_1 \ OP \ P_3)}$$

= [Except for Sequence, all compound operators are commutative ($\Box$, $\sqcap$, $[\![$ CS $]\!]$, $|||$)]

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP \in \{ \Box, \sqcap, [\![CS]\!], ||| \}}{(c_1 \models P_2 \ OP \ P_1) \rightsquigarrow (c_2 \models P_3 \ OP \ P_1)}$$

= [Compound Process Left 35]

true □

Sub-Rule 6.

***Sequence Compound Process Right**:*

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_3) \qquad OP = ;}{(c_1 \models P_1 \ OP \ P_2) \rightsquigarrow (c_2 \models P_1 \ OP \ P_3)}$$

In order to facilitate the proof, we will create and prove lemma C.10 and then prove sub-rule 6.

Proof

$(c_1 \models P_1 \ OP \ P_2) \rightsquigarrow (c_2 \models P_1 \ OP \ P_3)$

= [Lemma Compound Process C.10]

$c_1 \wedge c_2 \Rightarrow (\text{Lift } (gA\ (P_1)) \wedge \text{Lift } (gA\ (P_2)) \ ; P_1 \ OP \ P_2 \sqsubseteq (\text{Lift } (gA\ (P_1)) \wedge \text{Lift } (gA\ (P_2)) \ ; P_1 \ OP \ P_3))$

= [Be L = (Lift (gA (P_1)) $\wedge$ Lift (gA (P_2)))]

$c_1 \wedge c_2 \Rightarrow L \ ; P_1 \ OP \ P_2 \sqsubseteq L \ ; P_1 \ OP \ P_3$

= [Assumption OP = ;]

$c_1 \wedge c_2 \Rightarrow L \ ; P_1 \ ; P_2 \sqsubseteq (L \ ; P_1 \ ; P_3)$

= [$P_1 \sqsubseteq P_3$ (Assumption 5) $\wedge$ $P_2 \sqsubseteq P_2$, so P_1 ; $P_2 \sqsubseteq P_1$; P_3 (6)]

$c_1 \wedge c_2 \Rightarrow L \ ; P_1 \ ; P_2 \sqsubseteq L \ ; P_1 \ ; P_3$ [having $P_1 \ ; P_2 \sqsubseteq P_1 \ ; P_3$]

= [L $\sqsubseteq$ L $\wedge$ P_1 ; $P_2 \sqsubseteq P_1$; P_3, then (Monotonicity of Refinement 6) L ; P_1 ; $P_2 \sqsubseteq$ L ; P_1 ; P_3]

$c_1 \wedge c_2 \Rightarrow$ true

= (P $\Rightarrow$ true) = true

true

= true □

C.3.18 Hide Process

Attached Rule 37.

Hiding Advance:

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2)}{(c_1 \models P_1 \setminus S) \rightsquigarrow (c_2 \models P_2 \setminus S)}$$

From Assumptions C.19:

[Assumption 1]: $(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2)$
[Assumption 2]: Lift (getAssignments (P_1)) = Lift (getAssignments (P_2))
[Assumption 3]: Lift (getAssignments (P_1)) ; $P_1 \sqsubseteq$ Lift (getAssignments (P_2)) ; P_2)
[Assumption 4]: Lift (getAssignments (P_2)) ; P_1 Lift (getAssignments (P_2)) ; P_2
[Assumption 5]: $P_1 \sqsubseteq P_2$

Proof

$(c_1 \models P_1 \setminus S) \rightsquigarrow (c_2 \models P_2 \setminus S)$
= [Definition of Syntactic Transition C.27]
$\forall$ w . $c_1 \wedge c_2 \Rightarrow$
((Lift (getAssignments ($P_1 \setminus S$)) ; $P_1 \setminus S \sqsubseteq$ Lift (getAssignments ($P_2 \setminus S$)) ; $P_2 \setminus S$))
$\wedge$ Lift (getAssignments ($P_1 \setminus S$)) = Lift (getAssignments ($P_2 \setminus S$))
= [w is universally quantified, so we abstract it]
$c_1 \wedge c_2 \Rightarrow$
((Lift (getAssignments ($P_1 \setminus S$)) ; $P_1 \setminus S \sqsubseteq$ Lift (getAssignments ($P_2 \setminus S$)) ; $P_2 \setminus S$))
$\wedge$ Lift (getAssignments ($P_1 \setminus S$)) = Lift (getAssignments ($P_2 \setminus S$))
= [Definition of getAssignments (P $\setminus$ CS) C.1]
$c_1 \wedge c_2 \Rightarrow$
((Lift (getAssignments (P_1)) ; $P_1 \setminus S \sqsubseteq$ Lift (getAssignments (P_2)) ; $P_2 \setminus S$))
$\wedge$ Lift (getAssignments (P_1)) = Lift (getAssignments (P_2))
= [Assumption 2]
$c_1 \wedge c_2 \Rightarrow$
((Lift (getAssignments (P_1)) ; $P_1 \setminus S \sqsubseteq$ Lift (getAssignments (P_2)) ; $P_2 \setminus S$))
$\wedge$ true
= [P $\wedge$ true = P]
$c_1 \wedge c_2 \Rightarrow$
((Lift (getAssignments (P_1)) ; $P_1 \setminus S \sqsubseteq$ Lift (getAssignments (P_2)) ; $P_2 \setminus S$))
= [Assumption 2 again]

$c_1 \wedge c_2 \Rightarrow$
((Lift (getAssignments (P_1)) ; $P_1 \setminus S \sqsubseteq$ Lift (getAssignments (P_1)) ; $P_2 \setminus S$))
= [Assumption 5 and Hiding Monotonic C.2.9 $\Rightarrow$ $P_1 \setminus S \sqsubseteq P_2 \setminus S$. That with General Monotonicity of Refinement $\Rightarrow$ L ; P $\sqsubseteq$ L ; Q]
$c_1 \wedge c_2 \Rightarrow$ true
= [Predicate Calculus: P $\Rightarrow$ true = true]
$\forall$ w . true
=
true □

Attached Rule 38.

***Hiding Basic Process**:*
(c $\models$ *(***begin state** *ST PARS* • *A* **end***)* $\setminus$ *S)*
$\rightsquigarrow$
(c | *s* $\models$ **begin state** *ST PARS* • *(A* $\setminus$ *S)* **end***)*

Be HP = (**begin state** ST PARS • A **end**) $\setminus$ S

and

HB = **begin state** ST PARS • (A $\setminus$ S) **end**
Proof

$$\left(\begin{array}{l} (c \models (\textbf{begin state}\ ST\ PARS \bullet A\ \textbf{end}) \setminus S) \\ \rightsquigarrow \\ (c \mid s \models \textbf{begin state}\ ST\ PARS \bullet (A \setminus S)\ \textbf{end}) \end{array}\right)$$

= [Sigma Equal Basic Process 4 (having, from law C.2.19, that HP = HB]
true □

C.3.19 Rename Process

Attached Rule 39.

***Rename Advance**:*

$$\frac{(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2)}{(c_1 \models P_1\ [a_1, a_2, ... = b_1, b_2, ...]) \rightsquigarrow (c_2 \models P_2\ [a_1, a_2, ... = b_1, b_2, ...])}$$

From Assumptions C.19:

[Assumption 1]: $(c_1 \models P_1) \rightsquigarrow (c_2 \models P_2)$
[Assumption 2]: Lift (gA (P_1)) = Lift (gA (P_2))
[Assumption 3]: Lift (gA (P_1)) ; $P_1 \sqsubseteq$ Lift (gA (P_2)) ; P_2)
[Assumption 4]: Lift (gA (P_2)) ; P_1 Lift (gA (P_2)) ; P_2
[Assumption 5]: $P_1 \sqsubseteq P_2$

Proof

$(c_1 \models P_1\ [a_1, a_2, ... = b_1, b_2, ...]) \rightsquigarrow (c_2 \models P_2\ [a_1, a_2, ... = b_1, b_2, ...])$
= [Definition of Syntactic Transition C.27]

$$\forall w . c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} \left(\begin{array}{l} (Lift\,(gA\,(P_1\,[a_1, a_2, ... = b_1, b_2, ...]))\,;\ P_1\,[a_1, a_2, ... = b_1, b_2, ...]) \\ \sqsubseteq\ (Lift\,(gA\,(P_2\,[a_1, a_2, ... = b_1, b_2, ...]))\,;\ P_2\,[a_1, a_2, ... = b_1, b_2, ...]) \end{array} \right) \\ \wedge\ Lift\,(gA\,(P_1\,[a_1, a_2, ... = b_1, b_2, ...])) = Lift\,(gA\,(P_2\,[a_1, a_2, ... = b_1, b_2, ...])) \end{array} \right)$$

= [w is universally quantified, so we abstract it]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} \left(\begin{array}{l} (Lift\,(gA\,(P_1\,[a_1, a_2, ... = b_1, b_2, ...]))\,;\ P_1\,[a_1, a_2, ... = b_1, b_2, ...]) \\ \sqsubseteq\ (Lift\,(gA\,(P_2\,[a_1, a_2, ... = b_1, b_2, ...]))\,;\ P_2\,[a_1, a_2, ... = b_1, b_2, ...]) \end{array} \right) \\ \wedge\ Lift\,(gA\,(P_1\,[a_1, a_2, ... = b_1, b_2, ...])) = Lift\,(gA\,(P_2\,[a_1, a_2, ... = b_1, b_2, ...])) \end{array} \right)$$

= [Definition of gA (P [a_1, a_2, ... = b_1, b_2, ...]) C.1]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} \left(\begin{array}{l} (Lift\,(gA\,(P_1))\,;\ P_1\,[a_1, a_2, ... = b_1, b_2, ...]) \\ \sqsubseteq\ (Lift\,(gA\,(P_2))\,;\ P_2\,[a_1, a_2, ... = b_1, b_2, ...]) \end{array} \right) \\ \wedge\ Lift\,(gA\,(P_1)) = Lift\,(gA\,(P_2)) \end{array} \right)$$

= [Assumption 2]

$$c_1 \wedge c_2 \Rightarrow \left(\begin{array}{l} ((Lift\,(gA\,(P_1))\,;\ P_1\,[a_1, a_2, ... = b_1, b_2, ...]) \sqsubseteq (Lift\,(gA\,(P_2))\,;\ P_2\,[a_1, a_2, ... = b_1, b_2, ...])) \\ \wedge\ true \end{array} \right)$$

= [Predicate Calculus: P $\wedge$ true = P]

$c_1 \wedge c_2 \Rightarrow$
$((Lift\,(gA\,(P_1))\,;\ P_1\,[a_1, a_2, ... = b_1, b_2, ...] \sqsubseteq Lift\,(gA\,(P_2))\,;\ P_2\,[a_1, a_2, ... = b_1, b_2, ...]))$
= [Assumption 2 again]
$c_1 \wedge c_2 \Rightarrow$
$((Lift\,(gA\,(P_1))\,;\ P_1\,[a_1, a_2, ... = b_1, b_2, ...] \sqsubseteq Lift\,(gA\,(P_1))\,;\ P_2\,[a_1, a_2, ... = b_1, b_2, ...]))$
= [Assumption 5 and Rename Monotonic $\Rightarrow$ P_1 [a_1, a_2, ... = b_1, b_2, ...] $\sqsubseteq$ P_2 [a_1, a_2, ... = b_1, b_2, ...]. That with General Monotonicity of Refinement $\Rightarrow$ L ; P $\sqsubseteq$ L ; Q]
$c_1 \wedge c_2 \Rightarrow true$
= [Predicate Calculus: P $\Rightarrow$ true = true]
true □

Attached Rule 40.

Rename Basic Process:
(c $\models$ *(***begin state** *ST PARS* • *A* **end***) [*a_1*, *a_2*, ... = *b_1*, *b_2*, ...])*
$\rightsquigarrow$
(c | *s* $\models$ **begin state** *ST PARS* • *A [*a_1*, *a_2*, ... = *b_1*, *b_2*, ...]* **end***)*

Proof

Be

RP = (**begin state** ST PARS • A **end**) [a_1, a_2, ... = b_1, b_2, ...]

and

RB = **begin state** ST PARS • A [a_1, a_2, ... = b_1, b_2, ...] **end**

$$\left(\begin{array}{l} (c \models (\textbf{begin state}\ ST\,PARS \bullet A\ \textbf{end})\,[a_1, a_2, ... = b_1, b_2, ...]) \\ \rightsquigarrow \\ (c \mid s \models \textbf{begin state}\ ST\,PARS \bullet (A\,[a_1, a_2, ... = b_1, b_2, ...])\ \textbf{end}) \end{array}\right)$$

= [Sigma Equal Basic Process 4 (having, from law C.2.18, that RP = RB]

true □

C.3.20 Call Process

Attached Rule 41.

Parameterless Call Process:
(c ⊨ P) ⇝ (c ⊨ Content (P))

Proof (c ⊨ P) ⇝ (c ⊨ Content (P))
= [Definition C.8]
(c ⊨ Content (P)) ⇝ (c ⊨ Content (P))
= [Sigma Equal Sides 3]
true □

Attached Rule 42.

Call Process with normal parameters:
(c | s ⊨ P (N+)) ⇝ (c ⊨ ParamContent (P, [N+]))

Proof
(c ⊨ P) ⇝ (c ⊨ Content (P, [N+]))
= [Definition C.9]
(c ⊨ ParamContent (P, [N+])) ⇝ (c ⊨ ParamContent (P, [N+]))
true □

Attached Rule 43.

Call Process with indexed parameters:
(c ⊨ P ⌊ N+ ⌋) ⇝ (c ⊨ IndexedContent (P, [N+]))

Proof
(c ⊨ P ⌊ N+ ⌋) ⇝ (c ⊨ IndexedContent (P, [N+]))
= [Definition C.10]
(c ⊨ IndexedContent (P, [N+])) ⇝ (c ⊨ IndexedContent (P, [N+]))
= [Sigma Equal Sides 3]
true □

Attached Rule 44.

Call Process with generic parameters:
(c ⊨ P [N+]) ⇝ (c ⊨ GenericContent (P, [N+]))

Proof
(c ⊨ P [N+]) ⇝ (c ⊨ GenericContent (P, [N+]))

= [Definition C.11]
(c $\models$ GenericContent (P, [N+])) $\rightsquigarrow$ (c $\models$ GenericContent (P, [N+]))
= [Sigma Equal Sides 3]
true □

C.3.21 Iterated Process

Attached Rule 45.

***Iterated Processes**:*
(c $\models$ OP Decl • P) $\rightsquigarrow$ (c $\models$ IteratedExpansion (P, Decl, ITOPFLAG))
IteratedExpansion is defined on C.3.

Proof (c $\models$ OP Decl • P) $\rightsquigarrow$ (c $\models$ IteratedExpansion (P, Decl, ITOPFLAG))
= [Definition of Iterated Operator C.3]
(c $\models$ IteratedExpansion (P, Decl, ITOPFLAG)) $\rightsquigarrow$ (c $\models$ IteratedExpansion (P, Decl, ITOPFLAG))
= [Silent Transition Equal Sides 3]
true □

Bibliography

Abrial, J.-R. (1996), *The B-book: Assigning Programs to Meanings*, Cambridge University Press.

Alberto, Alex, Ana Cavalcanti, Marie-Claude Gaudel & Adenilso Simão (2017), 'Formal mutation testing for circus', *Information and Software Technology* **81**, 131–153.

Aljer, Ammar, Philippe Devienne, Sophie Tison, Jean-Louis Boulanger & Georges Mariano (2003), Bhdl: Circuit design in B, *em* 'ACSD', Citeseer, Pennsylvania, United States, pp. 241–242.

Barrocas, S. L. M. & M. Oliveira (2012), 'JCircus 2.0: an extension of the automatic translator from Circus to Java', *Communicating Process Architectures (CPA)* **.**(.).

Barrocas, S. L. M. & M. Oliveira (2016), 'A Validation Strategy for an Automatic Code Generator using Java Pathfinder', *Software Engineering Research and Practice (SERP)* **.**(.).

Barrocas, S. L. M. & M. Oliveira (2017), 'A Sound Operational Semantics for Circus', *Software Engineering Research and Practice (SERP)* **.**(.).

Barrocas, Samuel Lincoln Magalhaes (2011), JCircus 2.0: uma extensão da ferramenta de tradução de Circus para Java, Dissertaï¿½ï¿½o de mestrado, Universidade Federal do Rio Grande do Norte.

Bear, S. (1988), Structuring for the VDM Specification Language, *em* 'VDM 88', Dublin, Ireland.

Bicarregui, Juan, Matthew Bishop, Theodosis Dimitrakos, Kevin Lano, Tom Maibaum, Brian Matthews & Brian Ritchie (2000), 'Supporting co-use of vdm and b by translation', *Proceedings of VDM in* .

Börger, E. & R. F. Stärk (2003), *Abstract State Machines–A Method for High-Level System Design and Analysis*, Springer-Verlag.

Bossu, Gaelle & Antoine Requet (2000), 'Embedding formally proved code in a smart card: Converting b to c', *Formal Engineering Methods, International Conference on* **00**, 15.

Boudiaf, Noura & Abdelhamid Djebbar (2009), 'Towards an automatic translation of colored petri nets to maude language', *International Journal of Computer Science & Engineering* **3**(1).

Butler, M.J. & M. Leuschel (2005), Combining CSP and B for Specification and Property Verification, *em* 'FM', pp. 221–236.

Carl Hewitt, Peter Bishop, Richard Steiger (.), 'A Universal Modular ACTOR Formalism for Artificial Intelligence', . **.**, .

Cavalcanti, A. L. C., A. Sampaio & J. C. P. Woodcock (2005), 'Unifying classes and processes', *Journal of Software and Systems Modeling* **4**(3), 277–296.

Cavalcanti, Ana & Jim Woodcock (2006), 'A tutorial introduction to csp in unifying theories of programming', *Refinement techniques in software engineering* pp. 220–268.

Cavalcanti, Ana & Jim Woodcock (2013), CML Definition 2 - Operational Semantics (Deliverable Number: D23.3-4, Version 0.2), Relatï¿½rio tï¿½cnico, University of York.

Cavalcanti, Ana & Marie-Claude Gaudel (2010), Specification coverage for testing in circus., *em* 'UTP', Springer, pp. 1–45.

Cavalcanti, Ana & Marie-Claude Gaudel (2011), 'Testing for refinement in Circus', *Acta Inf.* **48**(2), 97–147.

ClearSy, Aix-en-Provence (2003), 'France. atelier b-user manual'.

Crocker, David (2003), 'Perfect developer: a tool for object-oriented formal specification and refinement', *Tools exhibition notes at formal methods Europe* .

Dantas, Bartira, David Déharbe, Stephenson Galvão, Anamaria Martins Moreira & Valério Medeiros Júnior (2009), 'Verified compilation and the B method: A proposal and a first appraisal', *Electr. Notes Theor. Comput. Sci.* **240**, 79–96.
URL: *http://dx.doi.org/10.1016/j.entcs.2009.05.046*

de Azevedo Oliveira, Diego (2017), BTestBox: uma ferramenta de teste para implementações B, Relatï¿½rio tï¿½cnico, Universidade Federal do Rio Grande do Norte.

de Matos, Ernesto Cid Brasil & Anamaria Martins Moreira (2012), BETA: A B based testing approach, *em* 'Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings', pp. 51–66.
URL: *http* : $//dx.doi.org/10.1007/978-3-642-33296-8_6$

Dijkstra, E. W. (1975), 'Guarded commands, nondeterminacy and the formal derivation of programs', *Communication of the ACM* **18**(18), 453–457.

Duran, Adolfo, Ana Cavalcanti & Augusto Sampaio (2010), 'An algebraic approach to the design of compilers for object-oriented languages', *Formal Asp. Comput.* **22**(5), 489–535.
URL: *http* : $//dx.doi.org/10.1007/s00165-009-0124-9$

Edmunds, Andrew, Michael J. Butler, Issam Maamria, Renato Silva & Chris Lovell (2012), Event-b code generation: Type extension with theories, *em* 'Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings', pp. 365–368.
URL: *http* : $//dx.doi.org/10.1007/978-3-642-30885-7_3 3$

Evans, Neil & Neil Grant (2008), 'Towards the formal verification of a java processor in event-b', *Electronic Notes in Theoretical Computer Science.* **201**, 45–67.

Fekete, Jean-Daniel & Martin Richard (1998), Esterel meets java: Building reactive synchronous programs in java, Relatï¿½rio tï¿½cnico, Technical Report Rapport 98-3-info, Ecole des Mines de Nantes.

Fischer, C. (1998), How to combine Z with a process algebra, *em* J. P.Bowen, A.Fett & M. G.Hinchey, eds., 'ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users', nï¿½mero 1493 *em* 'Lecture Notes in Computer Science', Springer Verlag, pp. 5–23.

Flanagan, Cormac & Patrice Godefroid (2005), Dynamic partial-order reduction for model checking software, *em* 'Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', POPL '05, ACM, New York, NY, USA, pp. 110–121.
URL: *http* : $//doi.acm.org/10.1145/1040305.1040315$

For (1999), *FDR: User Manual and Tutorial, version 2.82*.

Formiga, Andrei de A & Rafael D Lins (2009), Efficient implementation of the pi-calculus on the java virtual machine, *em* 'Third Workshop on Languages and Tools for Parallel and Distributed Programming LTPD 2009', p. 33.

Foster, Simon, Frank Zeyda & Jim Woodcock (2014), Isabelle/utp: A mechanised theory engineering framework, *em* 'Unifying Theories of Programming - 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers', Vol. 8963 de *Lecture Notes in Computer Science*, Springer, pp. 21–41.

Freitas, A. (2005), From *Circus* to Java: Implementation and Verification of a Translation Strategy, Dissertaï¿½ï¿½o de mestrado, Department of Computer Science, The University of York.

Freitas, Angela & Ana Cavalcanti (2006), 'Automatic translation from circus to java', *FM 2006: Formal Methods* pp. 115–130.

Freitas, Leo (2006), Model-Checking Circus, Tese de doutorado, Department of Computer Science, University of York. Leo Freitas.

Freitas, Leo (2008), *Circus*-LaTeX style explained - Community Z Tools (CZT), Technical report, Oxford University Computing Laboratory, Department of Computer Science, University of York, YO10 5DD, York, United Kingdom.

Galloway, A. & B. Stoddart (1997), *An operational semantics for ZCCS*, *em* M. G.Hinchey, ed., 'ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods', IEEE Computer Society, Washington, DC, USA, p. 272.

Garavel, Hubert & Joseph Sifakis (1990), Compilation and verification of lotos specifications, *em* 'PSTV', Vol. 10, pp. 359–376.

Gilbert, David R (1988), A lotos to parlog translator., *em* 'FORTE', pp. 31–44.

Gkatzia, Dimitra (2011), An investigation into the automated generation of provably correct code from formally verified designs.

Hentz, Cleverton, Jurgen J. Vinju & Anamaria M. Moreira (2015), *Reducing the Cost of Grammar-Based Testing Using Pattern Coverage*, Springer International Publishing, Cham, pp. 71–85.
URL: *https* : $//doi.org/10.1007/978-3-319-25945-1_5$

Hilderink, Gerald H., Jan F. Broenink, Wiek Vervoort & Andrè W. P. Bakkers (1997), Communicating Java Threads, *em* A. W. P.Bakkers, ed., 'Proceedings of WoTUG-20: Parallel Programming and Java', pp. 48–76.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice-Hall.

Hoare, C. A. R. & He Jifeng (1998), *Unifying Theories of Programming*, Prentice-Hall.

Hoffmann, Marc R. (2006), 'EclEmma'. At http://www.eclemma.org/.
URL: *http://www.eclemma.org/*

Hojjat, Hossein, Marjan Sirjani, Mohammad Reza Mousavi & Jan Friso Groote (2007), Sarir: A rebeca to mcrl2 translator, *em* 'Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), 10-13 July 2007, Bratislava, Slovak Republic', pp. 216–222.
URL: *http://doi.ieeecomputersociety.org/10.1109/ACSD.2007.62*

Horn, Alfred (1951), 'On sentences which are true of direct unions of algebras', *The Journal of Symbolic Logic* **16**(1), 14–21.

J. C. P. Woodcock, Ana Cavalcanti (2007), 'Operational semantics for circus', *Formal Aspects of Computing* . .

Jr, I. S. Medeiros, M. V. M. Oliveira & J. C. P. Woodcock (2012), Hard-wiring CSP Parallelism: Implementing Multi-Synchronisation and Interleaving, *em* D.Naumann & R.Gheyi, eds., '15th Brazilian Symposium on Formal Methods - Short Papers', Vol. 11, UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra Ã¯Â¿Â½ CCET, pp. 25 – 30. ISSN: 2178-6097.

Júnior, Valério Gutemberg Medeiros (2016), Método B e a síntese verificada para código de montagem, Tese de doutorado, Universidade Federal do Rio Grande do Norte, Brazil.

Kennedy, Andrew, Nick Benton, Jonas Braband Jensen & Pierre-Évariste Dagand (2013), Coq: the world's best macro assembler?, *em* '15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013', pp. 13–24.
URL: *http://doi.acm.org/10.1145/2505879.2505897*

Kimber, Tim G (2007), Object-Z to perfect developer, Tese de doutorado, Department of Computing, Imperial College London.

Kurshan, Robert, Vladimir Levin, Marius Minea, Doron Peled & Hüsnü Yenigün (1998), 'Static partial order reduction', *Tools and Algorithms for the Construction and Analysis of Systems* pp. 345–357.

Lämmel, Ralf & Wolfram Schulte (2006), Controllable combinatorial coverage in grammar-based testing, *em* 'IFIP International Conference on Testing of Communicating Systems', Springer, pp. 19–38.

Lattner, Chris (2002), LLVM: An infrastructure for multi-stage optimization, Tese de doutorado, Masterâs thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

Leroy, Xavier (2008), 'The compcert verified compiler, software and commented proof'.

Leroy, Xavier (2009), 'Formal verification of a realistic compiler', *Communications of the ACM* **52**(7), 107–115.

Leroy, Xavier (2014), Formal proofs of code generation and verification tools, *em* 'Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings', pp. 1–4.
URL: *http* : $//dx.doi.org/10.1007/978-3-319-10431-7_1$

Leroy, Xavier (2016), The CompCert C verified compiler: Documentation and userâs manual, Tese de doutorado, Inria.

Leuschel, Michael (2008), Towards demonstrably correct compilation of java byte code, *em* 'FMCO', Springer, Berlin, Germany, pp. 119–138.

Li, Liwu (2005*a*), 'An implementation of the pi-calculus on the .net', *Journal of Object Technology* **4**(5), 20.

Li, Liwu (2005*b*), 'Implementing the p-calculus in java.', *Journal of Object Technology* **4**(2), 157–178.

Liu, Shaoying & Shin Nakajima (2011), A "vibration" method for automatically generating test cases based on formal specifications, *em* 'Software Engineering Conference (APSEC), 2011 18th Asia Pacific', IEEE, pp. 73–80.

M. G. Hinchey and S. A. Jarvis (1995), *Concurrent Systems: Formal Development in CSP*, McGraw-Hill, Inc., New York, NY, USA.

Macário, F. J. S. & M. V. M. Oliveira (2015), Hard-wiring CSP Hiding: Implementing Channel Abstraction to Generate Verified Concurrent Hardware, *em* M.Cornélio & B.Roscoe, eds., 'Formal Methods: Foundations and Applications - 18th Brazilian Symposium on Formal Methods', Vol. **9526** de *Lecture Notes in Computer Science*, Springer-Verlag, pp. 3 – 18. The original publication is available at www.springerlink.com.
URL: *http* : $//dx.doi.org/10.1007/978-3-319-29473-5_1$

Malik, P. & M. Utting (2005), CZT: A Framework for Z Tools, *em* H.Treharne, S.King, M. C.Henson & S. A.Schneider, eds., 'ZB', Vol. **3455** de *Lecture Notes in Computer Science*, Springer, pp. 65–84.

Mammar, Amel & Régine Laleau (2006), 'From a B formal specification to an executable code: application to the relational database domain', *Information & Software Technology* **48**(4), 253–279.
URL: *http://dx.doi.org/10.1016/j.infsof.2005.05.002*

Mandrioli, Dino, Sandro Morasca & Angelo Morzenti (1995), 'Generating test cases for real-time systems from logic specifications', *ACM Transactions on Computer Systems (TOCS)* **13**(4), 365–398.

Medeiros Junior, Ivan Soares de (2012), Geração automática de hardware a partir de especificações formais: estendendo uma abordagem de tradução, Dissertaï¿½ï¿½o de mestrado, Universidade Federal do Rio Grande do Norte.

Méry, Dominique & Neeraj Kumar Singh (2010), Eb2c: A tool for event-b to c conversion support, *em* '8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)'.

Méry, Dominique & Neeraj Kumar Singh (2011), Automatic code generation from event-b models, *em* 'Proceedings of the Second Symposium on Information and Communication Technology', SoICT '11, ACM, New York, NY, USA, pp. 179–188.
URL: *http://doi.acm.org/10.1145/2069216.2069252*

Miyazawa, Alvaro & Ana Cavalcanti (2016), 'Scj-circus: a refinement-oriented formal notation for safety-critical java', *arXiv preprint arXiv:1606.02021* .

Miyazawa, Alvaro Heiji (2008), Geração parcial de código Java a partir de especificações formais Z., Tese de doutorado, Universidade de São Paulo.

Morgan, C. & P. H. B. Gardiner (1990), 'Data refinement by calculation', *Acta Informatica* **27**(6), 481–503.

Murali, Rajiv & Andrew Ireland (2012), 'E-spark: Automated generation of provably correct code from formally verified designs', *Electronic Communications of the EASST* **53**.

NASA (2005), 'Java Pathfinder'. At http://javapathfinder.sourceforge.net/.
URL: *http://javapathfinder.sourceforge.net/*

Nogueira, Sidney, Augusto Sampaio & Alexandre Mota (2008), 'Guided test generation from csp models', *Theoretical Aspects of Computing-ICTAC 2008* pp. 258–273.

Oliveira, M, A Sampaio, PRG Antonino, RT Ramos, A Cavalcanti & J Woodcock (2013), 'Compositional analysis and design of cml models', *COMPASS Deliverable D* **24**.

Oliveira, M. V. M. (2006*a*), Formal Derivation of State-Rich Reactive Programs using *Circus*, Tese de doutorado, Department of Computer Science, University of York. Oliveira.

Oliveira, M. V. M. (2006*b*), Formal Derivation of State-Rich Reactive Programs using *Circus* - Extended Version, Tese de doutorado, Department of Computer Science, University of York. Oliveira.

Oliveira, M. V. M., A. C. Gurgel & C. G. de Castro (2008), CRefine: Support for the *Circus* Refinement Calculus, *em* A.Cerone & S.Gruner, eds., '6th IEEE International Conferences on Software Engineering and Formal Methods', IEEE Computer Society Press, pp. 281–290.

Oliveira, M. V. M., I. S. Medeiros Júnior & J. C. P. Woodcock (2013), A verified protocol to implement multi-way synchronisation and interleaving in csp, *em* R. M.Hierons, M. G.Merayo & M.Bravetti, eds., 'Software Engineering and Formal Methods', Vol. 8137 de *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 46–60.
URL: $http://dx.doi.org/10.1007/978-3-642-40561-7_4$

Oliveira, Marcel Vinicius Medeiros, Augusto C. A. Sampaio & Madiel S. Conserva Filho (2014), Model-checking circus state-rich specifications, *em* 'Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings', pp. 39–54.
URL: $http://dx.doi.org/10.1007/978-3-319-10181-1_3$

Palsberg, Jens (1992*a*), An automatically generated and provably correct compiler for a subset of ada, *em* 'ICCL'92, Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, USA, 20-23 Apr 1992', pp. 117–126.
URL: *http://dx.doi.org/10.1109/ICCL.1992.185474*

Palsberg, Jens (1992*b*), A provably correct compiler generator, *em* 'ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings', pp. 418–434.
URL: *http* : $//dx.doi.org/10.1007/3-540-55253-7_25$

Peled, Doron (1998), Ten years of partial order reduction, *em* 'Computer Aided Verification', Springer, pp. 17–28.

Phillips, Jonathan D. & G. S. Stiles (2004), 'An Automatic Translation of CSP to Handel-C', *Formal Aspects Of Computing* **.**(.), 19 – 38.

Raju, V., L. Rong & G. S. Stiles (2003), Automatic conversion of csp to ctj, jcsp, and ccsp, *em* 'In'.

Robin Milner, Joachim Parrow, David Walker (1990), *A Calculus for mobile processes*, ., .

Sampaio, Augusto (1997), *An Algebraic Approach to Compiler Design*, Vol. 4 de *AMAST Series in Computing*, World Scientific.
URL: *http* : $//dx.doi.org/10.1142/2870$

Spivey, J. M. (1992), *The Z Notation: A Reference Manual*, 2ndï¿½ ediï¿½ï¿½o, Prentice-Hall.

Tannen, Val (2009), First-order logic: Syntax, *em* 'Encyclopedia of Database Systems', Springer, pp. 1139–1141.

Tripakis, Stavros, Christos Sofronis, Paul Caspi & Adrian Curic (2005), 'Translating discrete-time simulink to lustre', *ACM Transactions on Embedded Computing Systems (TECS)* **4**(4), 779–818.

V. Raju, L. Rong & G. S. Stiles (2003), 'Automatic Conversion of CSP to CTJ', *Communicating Process Architectures* **.**(.), 63 – 81.

Vargas, Abigail Parisaca, Lizeth Tapia & Chris George (2008), A translation from rsl to csp, *em* 'Chilean Computer Science Society, 2008. SCCC'08. International Conference of the', IEEE, pp. 119–126.

Vlissides, John, Richard Helm, Ralph Johnson & Erich Gamma (1995), 'Design patterns: Elements of reusable object-oriented software', *Reading: Addison-Wesley* **49**(120), 11.

Welch, P. H. (2000), Process Oriented Design for Java. Concurrency for All, *em* 'Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications', Vol. **1**, pp. 51 – 57.

Welch, Peter H., Neil C. C. Brown, James Moores, Kevin Chalmers & Bernhard H. C. Sputh (2010), 'Alting Barriers: Synchronisation with Choice in Java using JCSP', *Concurrency and Computation: Practice and Experience* **22**, 182–196. The DOI should redirect to http://onlinelibrary.wiley.com.chain.kent.ac.uk/doi/10.1002/cpe.1471/abstract .

URL: *http* : *//www.cs.kent.ac.uk/pubs/*2010/3068

Woodcock, J. C. P. & A. L. C. Cavalcanti (2001), A concurrent language for refinement, *em* A.Butterfield & C.Pahl, eds., 'IWFM'01: 5th Irish Workshop in Formal Methods', BCS Electronic Workshops in Computing, Dublin, Ireland.

Woodcock, J. C. P. & A. L. C. Cavalcanti (2004), A Tutorial Introduction to Designs in Unifying Theories of Programming, *em* E. A.Boiten, J.Derrick & G.Smith, eds., 'IFM 2004: Integrated Formal Methods', Vol. **2999** de *Lecture Notes in Computer Science*, Springer-Verlag, pp. 40–66. Invited tutorial.

Woodcock, J. C. P. & J. Davies (1996), *Using Z—Specification, Refinement, and Proof*, Prentice-Hall.

Wright, Steve (2009), Automatic generation of c from event-b, *em* 'Workshop on integration of model-based formal methods and tools', Citeseer, p. 14.

Xavier, M. A., A. L. C. Cavalcanti & A. C. A. Sampaio (2006), Type Checking *Circus* Specifications, *em* A. M.Moreira & L.Ribeiro, eds., 'SBMF 2006: Brazilian Symposium on Formal Methods', pp. 105 – 120.

Xavier, Manuela Almeida (2006), Definição e Implementação do Sistema de Tipos da Linguagem Circus, Dissertaï¿½ï¿½o de mestrado, Universidade Federal de Pernambuco.