Formal Derivation of State-Rich Reactive Programs using *Circus*

Marcel Vinícius Medeiros Oliveira

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

2005

To Lauana and my parents

Abstract

The lack of formalism in most software developments can lead to a loss of precision and correctness in the resulting software. Formal techniques of program development have been developed in the past decades and can tackle this problem. Two different approaches have been taken: one focuses on data aspects, and the other focuses on behavioural aspects of the systems. Some combined languages have already been proposed to integrate these two approaches; however, as far as we know, none of them, apart from *Circus*, includes a refinement calculus.

This work presents a method that can be applied in order to achieve a formal derivation of state-rich reactive programs, using *Circus*, in a calculational style. It extends the existing *Circus* refinement strategy to reach Java implementation and formalises its refinement calculus. For that we proposed and mechanised a denotational semantics for *Circus*, which was used to prove over one-hundred and forty refinement laws, many of which were suggested by an industrial case study on which we worked. As far as we know, this is the only mechanised semantics for languages like *Circus*; the mechanisation of the *Circus* semantics and its theoretical basis, the *Unifying Theories of Programming*, are the basis for a theorem prover for *Circus*. A translation strategy from *Circus* to Java is also an important part of this work.

Our method is illustrated by the case study: a safety-critical fire control system. So far, this is the largest case study on the *Circus* refinement calculus. We present the refinement of its abstract centralised specification to a concrete distributed one, and then its translation to Java, using our translation strategy. We believe that this industrial case study provides empirical evidence that the formal development of state-rich reactive processes using *Circus* is possible in both theory and practice.

Contents

List of Figures

List of Tables

List of Accompanying Material

This document is accompanied by a CD, which contains the electronic version of this document and the following additional material.

- ☞ **Formal Derivation of State-Rich Reactive Programs using *Circus* - Extended Version**: extended version of this document containing the full specification and derivation of our case study presented in Chapter 5; the full definition of our translation strategy to Java; the translation of the process *Summation* used to illustrate the strategy in Chapter 6; and the proofs of the refinement laws.
- ☞ **ProofPower-Z Theories Documentation**: documentation of the theories created in the mechanisation of the UTP theories and *Circus* in ProofPower-Z, presented in Chapter 3.
- ☞ **ProofPower-Z Theories Source Code**: executable source of the theories documented in the item above.
- ☞ **Fire Control System Source Code**: Java source code of our case study presented in Chapter 5.
- ☞ **Summation Source Code**: Java source code of the *Summation* example used in Chapter 6.
- ☞ **Chronometer Source Code**: Java source code of the *Chronometer* example used in Chapter 1.

Alternatively, this material can be downloaded from the following URL:

➲ `http://www.cs.york.ac.uk/circus/refinement-calculus/oliveira-phd/`

Acknowledgments

> ❝ **If I have been able to see further, it was only because I stood on the shoulders of giants.** ❞ ✍ Sir Isaac Newton

This thesis is the result of a long term collaboration with my supervisor, Ana Cavalcanti, who in so many occasions comforted, inspired and encouraged me to take the route that lead me to this point. Jim Woodcock was also another great source of enlightening. Their integrity, dedication, and friendship inspired me throughout my doctorate. The final result of this thesis was immensely benefited by their insights.

The work with ProofPower had a large support from Roger Bishop Jones, Rob Arthan, Mark Adams and Philip Clayton. The technical nuances that were raised during this part of the thesis were solved with their patience and help; furthermore, the proving techniques used were inspired by them. Will Harwood also provided valuable advice for this work: the definition of the language constructors as functions is due to him. During the development of the translation strategy from *Circus* to JCSP, discussions and opinions were exchanged with Alistair McEwan and Peter Welch. Besides, the validation and mechanisation of the translation strategy done by Angela Freitas suggested some corrections that improved its final version presented in this thesis. The reviews of Rogério de Lemos, Eerke Boiten, and anonymous committee members, have also contributed to this final version of the thesis. The countless exchange of ideas and opinions with Adnan Sherif, Adolfo Duran, Alexandre Mota, Augusto Sampaio, Diyaa-Addein Atiya, Jeremy Jacob, Leonardo Freitas, Lindsay Groves, Rodolfo Gomez, Steve King, and Xinbei Tang was yet another source of inspiration for this thesis. I am also very grateful for the financial support from QinetiQ and the Royal Society.

To my friends in Brazil, thanks very much for their words of support in a large number of occasions. To the new friends I made in England, especially Adolfo, Nícia, Renato, Viviane, Osmar and Simone, with whom I shared some of my best moments.

The unconditional support given by my parents, brother, and sisters, not only during my time in England but throughout my very existence, and their unstoppable encouragement, made it possible for me to get to this point.

Days and sleepless nights, happy and sad moments, difficulties, worries, they were all shared with the most lovely and caring wife one can have: Lauana. She was always there for me and made me look on the bright side of things, strengthening my faith in the plans that God has for my life. To her, my eternal gratitude and love.

Hosanna to the Lord, for putting all these wonderful people in my way to comfort, inspire, encourage, enlighten, support, love and care for me. Praised be His name.

Canção do Exílio
✍ ***Gonçalves Dias***

Minha terra tem palmeiras,
Onde canta o Sabiá;
As aves, que aqui gorjeiam,
Não gorjeiam como lá.

Nosso céu tem mais estrelas,
Nossas várzeas têm mais flores,
Nossos bosques têm mais vida,
Nossa vida mais amores.

Em cismar, sozinho, à noite,
Mais prazer encontro eu lá;
Minha terra tem palmeiras,
Onde canta o Sabiá.

Minha terra tem primores,
Que tais não encontro eu cá;
Em cismar sozinho, à noite
Mais prazer encontro eu lá;
Minha terra tem palmeiras,
Onde canta o Sabiá.

Não permita Deus que eu morra,
Sem que eu volte para lá;
Sem que desfrute os primores
Que não encontro por cá;
Sem quinda aviste as palmeiras,
Onde canta o Sabiá.

Declaration

I hereby declare that the contents of this thesis are the result of my own original contribution, except where otherwise stated. I have acknowledged other sources of joint work through explicit referencing. The following material, presented in this thesis, has been previously published:

[1] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying Theories in ProofPower-Z. In *First International Symposium on Unifying Theories of Programming*, *LNCS*. Springer-Verlag, 2006. To Appear.

[2] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Formal development of industrial-scale systems. *Innovations in Systems and Software Engineering—A NASA Journal*, **1**(2):125–146, 2005.

[3] M. V. M. Oliveira and A. L. C. Cavalcanti. From *Circus* to JCSP. In J. Davies *et al.*, editor, *Sixth International Conference on Formal Engineering Methods*, volume **3308** of *LNCS*, pages 320–340. Springer-Verlag, November 2004.

[4] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining industrial-scale systems in *Circus*. In Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures*, volume **62** of *Concurrent Systems Engineering Series*, pages 281–309. IOS Press, 2004.

[5] M. V. M. Oliveira. *A Refinement Calculus for Circus - Mini-thesis.* Technical Report 8-04, University of Kent, April 2004.

Marcel Vinícius Medeiros Oliveira
✉ marcel@cs.york.ac.uk
☎ +44 (0)1904 433244
➲ http://www.cs.york.ac.uk/~marcel/

Chapter 1

Introduction

In this chapter, we present the motivations for the development of our work. Furthermore, we discuss the objectives of our work, and illustrate the use of the *Circus* refinement calculus with a simple example. Finally, an overview of this document is presented.

1.1 Motivation

The current lack of formalism in most software developments raises difficulties in developing relatively low cost trustworthy software within a well-defined and controllable time frame. Previous experience with the informal techniques, which resulted in the software crisis, was the main motivation for the start of the use of formal methods in the development processes of safety-critical systems. By stressing the importance of a rigorous semantics for the notation used, the use of formal methods allows us to achieve a depth in the analysis of computing systems that would otherwise be impossible.

In fact, software industry leaders, like Microsoft, have already noticed this need for formal methods, and currently invest a considerable amount of their resources in the development of formal verification technologies [12]. By following a formal software process consistently, these companies may achieve better quality products, more efficient teams and individuals, reduced costs, and better morale.

One can summarise the formal development of systems in terms of two approaches; both of them start at a formal (abstract) specification. In the first approach, we propose a formalisation for a subsequent design and then verify it against the abstract [42]. In the *correct-by-construction* approach [36, 100], the design is gradually calculated as the result of incremental manipulation of the specification using refinement laws; these may reduce nondeterminism, as well as introduce executable constructs. Both approaches are complementary and useful in practice; we focus our attention on the second one.

The availability of a refinement calculus provides us with the possibility of correctly constructing programs in a stepwise fashion. Each step is justified by the application of a refinement law (possibly with the discharge of proof obligations). Together, the refinement laws provide us with a framework for the construction process. This derives from the fact that only valid laws can be applied at a certain time.

Throughout the past decades two schools have been developing formal techniques for precise, correct, and concise software development. However, they have taken different approaches: one of them has focused on data aspects of the system, while the other one has focused on the behavioural aspects of the system.

Languages like Z [107], VDM (Vienna Development Method) [57], ASM (Abstract State Machines) [17], and B [3], use a model-based approach, where mathematical objects from set theory form the basis of the specification. Although possible in a rather difficult and implicit fashion, specification constructs to model behavioural aspects such as choice, sequence, parallel composition, and others, are not explicitly provided by any of these languages.

In [29], a refinement calculus for Z (ZRC) is presented. This work is based on Morgan's work [63], and incorporates the Z notation, following his style and conventions. In VDM, rules for data and operation refinement allow one to establish links between abstract requirements specifications and detailed design specifications down to the level of code. A refinement calculus for ASM is presented in [78] and gives support for the development of ASM specifications. Finally, the B-Method provides verification methods for refinement, which are supported by the B-Toolkit [7].

On the other hand, CSP (Communicating Sequential Processes) [52, 80] and CCS (Calculus of Communicating Systems) [62], among others, provide constructs that can be used to describe the behaviour of the system. However, they do not support a concise and elegant way to describe the data aspects of the system. In [80], three different notions of refinement are presented: traces refinement, stable-failures refinement, and failures-divergences refinement, which are supported by tools like FDR [43].

The combination of different formalisms allows the *reuse of notations* in an integrated framework that is able to describe different aspects of the systems. Some of these combinations have taken a syntactic approach [40], in which one formalism is embedded into another, and the choices of which formalism to combine were based on the availability of *tool support* and the possibility of *reusing* these *tools*. However, restrictions on the *architecture* of the systems have to be made in order to achieve this. On the other hand, the semantic approach, in which different formalism are combined in a common semantic model, needs direct tool support for the semantic basis and, as a consequence, for the combined notation. As a matter of fact, there is a wide spectrum of choices for combinations and a trade-off has to be made between availability of tools and the convenience of the combinations.

Two other aspects that ought to be considered are modularity of specifications and, more importantly, compositionality of refinement. In a syntactic combination, refinement of the different aspects of the system is done separately: the refinement of one of the parts is the refinement of the whole system; this also requires restrictions on the architecture of the system. On the other hand, any integrated semantics, as the one I present in this thesis, allows us to formalise an integrated refinement calculus, in which refinement of concurrent and data aspects of the system can be done in the same context: sequential refinement can be done in the context of concurrency with no restrictions whatsoever on the system's architecture and vice-versa. Furthermore, a full integration (freely mixed) of concurrency and data aspects allows us to reach more efficient and less complicated implementable code.

Many formalisms combine data and behavioural aspects of the system. Z has been used as the basis of a calculus for communicating state machines [92]. Combinations of Z with CCS [46, 47, 93], Z with CSP [40, 82, 66], Object-Z [21] with CSP [39, 87, 60], and Object-Z with timed CSP [60] are some of these attempts to combine both schools. Furthermore, combinations of B and action systems [4], B and CSP [95], and notations that describe both aspects, like RAISE [50], the Rigorous Approach to Industrial Software Engineering, have been used. However, as far as we know, none of them has a related refinement calculus. Furthermore, apart from Fischer's CSP-OZ (Java) and RAISE (C++ and Ada), none of these works provide a strategy of translation into implementable code. Lai and Sanders propose a refinement calculus for communicating processes with states [59]. They extend an occam-like language with a specification statement in the style of Morgan [63]. Unfortunately, the operators allowed in this language are limited and no data refinement method is proposed.

The lack of support for refinement of state-rich reactive systems in a calculational style as that presented in [63] has motivated the creation of *Circus* (Concurrent Integrated Refinement CalculUS) [102, 103]. In this concurrent language, systems are characterised as processes, which group constructs that describe data and control behaviour; the Z notation [91] is used to define most of the data aspects, and CSP is used to define behaviour. Besides, the language provides support for formal stepwise development of concurrent programs [84, 26, 27].

Predicate transformers [37] are commonly used as the basis of semantic models for imperative refinement calculi [8, 65, 63]. However, a different model is used as the basis of theories of refinement for CSP, the failures-divergence model [52, 80]. Other works, such as those presented in [87, 39], provide a failures-divergences model for Object-Z classes, in order to present the semantics for combinations of Object-Z and CSP. Although data refinement was investigated for these combinations, no refinement laws were proposed. In [108], the failures model has been used to give behavioural semantics to abstract data types. In order to be able to give a semantics to *Circus*, we need to use a semantic model that is able to combine the notions of refinement for CSP and for imperative programs. The *Unifying Theories of Programming* (UTP) [54] is a framework that makes this combination possible by unifying programming science across many different computational paradigms.

In [105], Cavalcanti and Woodcock present a semantic model for *Circus* based on the UTP. Although usable for reasoning about systems specified in *Circus*, it is not appropriate to prove properties of *Circus* itself (e.g. our refinement laws). This happens because a shallow embedding, in which the mapping from *Circus* constructs to their semantic representation as a Z specification, with yet another language being used as a meta-language, would not allow us to express these laws. For this reason, a new semantics, which allows us to reason about the refinement laws, must be given to *Circus*.

In [27], a refinement strategy for *Circus*, as well as some refinement laws, was presented. However, the verification of these laws, the proposition of a comprehensive set of refinement laws, and further case studies were left as future work. The existence of tool support for refinement is an important piece of work that makes the development of systems using *Circus* a reality in practice. For this reason, a mechanisation of the semantics and of the *Circus* refinement calculus is needed.

Finally, the result of refining a *Circus* specification is a program written in a combination of CSP and guarded commands. In order to implement this program, we still need a link between *Circus* and a practical programming language.

1.2 Objectives

This work provides and formalises a cost-effective method of formal derivation for state-rich reactive programs using *Circus*. In [27], Cavalcanti *et al.* present a case study on the *Circus* refinement strategy: a reactive buffer. The authors present the refinement from an abstract specification to a concrete one. I intend to go further in my strategy and case study: support and illustrate refinement from an abstract specification to Java code.

The *Circus* semantics presented in [103] does not allow us to prove the refinement laws. For this reason, we need to redefine the *Circus* semantics as a deep embedding of *Circus* in Z. In this approach, the syntax and the semantics of *Circus* is formalised in Z. Based on the semantics presented here, we prove over ninety percent of the one-hundred and forty-six proposed refinement laws. These proofs range over all the structure of the language and include all the data simulation laws. This involves the proof of one-hundred and ten theorems, two-hundred and eighteen auxiliary lemmas, and one-hundred and thirty-three refinement laws. We present some of these proofs in this document; the extended version of this thesis [71] contains all the remaining proofs.

Since the semantic model used for the *Circus* semantics is based on the UTP, before mechanising *Circus* itself, we have to mechanise the UTP theories that give basis to it. This work consists in the construction of the theories of relations, designs, reactive processes, and CSP processes. Besides, we include over four-hundred and seventy theorems related to these theories. The mechanisation of these theories enables a further exploration of the results presented in [31], where the authors summarise the alphabetised relational calculus, and the theory of pre-postcondition specifications, called designs, present a detailed theory for reactive processes, and then combine it with the theory of designs to provide the model for CSP. Our work provides the basis of a theorem prover for *Circus* by mechanising its semantics. The mechanisation of a state-rich reactive language like *Circus* is yet another novel contribution to the field.

In order to make these results as general as possible, we have created a theory hierarchy that allows users to inherit from theories that they really intend to use. Our mechanisation of the UTP provides mechanical support not only for the *Circus* semantics, but also for all the languages based on the UTP.

Some issues were raised during the mechanisation of the UTP and *Circus* theories. An important choice was to represent syntax as functions, which allows us to extend the language without the need to prove any previously proved theorem again. Some other issues arose from the existence of an alphabet in the UTP. The difference between variables values and names is not explicit in [54], but important for the mechanisation. This was the reason for creating a set-based model for predicates, instead of using the standard predicate calculus already existent in the theorem prover, even though it meant that we had to prove the laws of the predicate calculus in this new model.

The choice of the theorem prover for the mechanisation of *Circus* and its refinement calculus was not an issue. The large number of formally verified theories, including elementary number theory, algebra, set theory, linear arithmetics, and many Z related theories, was one of the reasons of the choice for ProofPower-Z [1] as the theorem prover used in the mechanisation of the *Circus* refinement calculus. Furthermore, *Circus* is largely funded by Qinetiq, who routinely use ProofPower-Z, and intend to use *Circus* in their development process. This theorem prover was indeed a natural choice as a basis for the mechanisation of *Circus* and its refinement calculus.

In order to verify the usefulness of the set of laws proposed in [27], a more significant case study on the refinement of *Circus* programs must be taken into account [75, 76]. In this thesis, we present an industrial case study on the *Circus* refinement calculus: a safety-critical fire control system. As far as we know, it is the largest case study on the *Circus* refinement calculus. The transformation of an abstract centralised specification of this system to the Java implementation of a distributed one is in the scope of this work.

Throughout the development of our case study there were some problems; we present their solutions in this thesis. First, the set of laws presented in [27] was not sufficient; we propose new refinement laws (marked in Appendix C with a *). For instance, we require some laws for inserting and distributing assumptions, and a new process refinement law. In total, almost one-hundred new laws have been identified during the development of our case study.

In [27], the refinement of mutually-recursive actions is not considered; our case study, however, includes mutually recursive definitions. We present here a notation used to prove refinement of such systems; this results in more concise and modular proofs. The proofs of the necessary theorems that justify the notation are also part of our work.

The case study illustrates an application of the refinement strategy in an existing industrial application. We believe that, with the results in Chapter 5, we provide empirical evidence of the power of expression of *Circus* and, principally, that the strategy presented in [27] is applicable to large industrial systems.

The final contribution of this thesis is a link between *Circus* and a practical programming language, Java. This translation strategy is based on a number of translation rules, which, if applied exhaustively, transform a *Circus* program into a Java program that uses the JCSP library [99]. These rules capture and generalise the approach that we took in the implementation of our case study.

The strategy is applicable to programs written in an executable subset of *Circus*. We assume that, before applying the translation strategy, the specification of the system we want to implement has been already refined to a *Circus* specification that uses only constructors of this subset, using the *Circus* refinement strategy presented in Chapter 4.

The existence of tool support for refinement and automated translation to Java makes formal development based on *Circus* relevant in practice. Such a systematic strategy can be used as a guideline for implementing *Circus* programs, and we do implement our case study in Java. Furthermore, the translation strategy was used as a guideline for mechanising the translation of *Circus* programs to Java [44]. However, the rules presented here still needed to be proved. Currently, we rely on the validation of the strategy during the implementation of our case study, and many other examples, and on the rather direct correspondence between *Circus* and JCSP; a step towards the formal validation of these rules is presented in [44].

In summary, by the end of this document, we intend to have provided enough support for the following proposition:

Thesis Proposition

***Circus* can be given a refinement calculus, which is sound and applicable to industrial safety-critical state-rich reactive systems. Furthermore, the derivation of these systems from an abstract specification into implementable code can be formalised.**

In what follows, we start by illustrating our method with the development of a simple example: a chronometer.

1.3 A Simple Development

The starting point for any *Circus* program development is an abstract (usually centralised) specification of the system. Using the refinement strategy presented in Chapter 4, we can transform this abstract specification into a concrete (usually distributed) specification of the system. This refinement strategy is based on refinement iterations that may include three steps: simulation, action refinement, and process refinement. The first two steps reorganise the internal structure of the process, by introducing the elements of the concrete system state, and refining the actions into two partitions in a way such that each partition operates on different components of the modified state. After this, the process is actually partitioned: each partition clearly has a independent state and behaviour. The third step upgrades each of these partitions to individual processes: we combine the resulting processes in the same way as their main actions were in the original process. We apply as many iterations as needed until we have an implementable *Circus* specification. At this point we apply the translation strategy from *Circus* to Java presented in Chapter 6; this results in Java code that implements the system that was initially specified.

Our example chronometer interacts with the environment via three channels. It receives a *tick* signal every second, and if asked about its current time via channel *time*, it outputs in channel *out* a pair of numbers $(minutes, seconds)$ ranging within the type $RANGE == 0..59$. Channels are declared using the keyword **channel**; we declare the name of the channel and the type of the values it can communicate.

$$\mathbf{channel}\ tick, time$$
$$\mathbf{channel}\ out : RANGE \times RANGE$$

The abstract process *AChronometer*, whose definition is given below between the keywords **begin end**, has both the *sec*onds and the *min*utes of its current counting as its state components. The **state** is declared using a Z schema, as presented below. For conciseness,

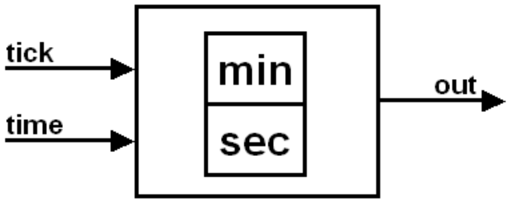


Figure 1.1: Abstract Chronometer

some schemas may be presented in their horizontal form $name \mathrel{\widehat=} [\, decl \mid pred \,]$.

$$\begin{array}{l} \textbf{process}\ AChronometer \mathrel{\widehat=} \\ \quad \textbf{begin state}\ AState \mathrel{\widehat=} [\, sec, min : RANGE \,] \end{array}$$

The state is initialised by the Z operation $AInit$, which sets both components to zero.

$$AInit \mathrel{\widehat=} [\, AState' \mid sec' = min' = 0 \,]$$

Undashed variables represent the values of the variables before the execution of an operation; on the other hand, dashed variables represent the values of the variables after the execution of an operation. The decoration of a schema, for instance $Schema'$, where $Schema \mathrel{\widehat=} [\, x_1 : T_1 \ldots\ x_n : T_n \mid p \,]$ is a new schema whose components are obtained by applying the decoration to all the components of the original schema; and the modification of the predicate part of the new schema reflects the new names of the components. For instance, we have that $Schema' \mathrel{\widehat=} [\, x_1' : T_1 \ldots\ x_n' : T_n \mid p\,[x_1'/x_1, \ldots\ , x_n'/x_n] \,]$. Finally, the inclusion of the schema $AState'$ in the declaration part of $AInit$ merges the declarations of both schemas, and conjoins their predicates.

Seconds are incremented by one in the $IncSec$ operation; however, every sixty seconds the seconds are set to zero, since the chronometer will start counting another minute.

$$IncSec \mathrel{\widehat=} [\, \Delta AState \mid sec' = (sec + 1) \mathsf{\ mod\ } 60 \,]$$

For any schema $Schema$, $\Delta Schema$ is a schema that includes both $Schema$ and $Schema'$.

The Z operation $IncMin$ increments the minutes. For conciseness, we consider that our chronometer counts only seconds and minutes. As for the seconds, the chronometer's minutes value resets every sixty minutes.

$$IncMin \mathrel{\widehat=} [\, \Delta AState \mid min' = (min + 1) \mathsf{\ mod\ } 60 \,]$$

The CSP action Run represents the execution of a cycle of the abstract chronometer. If it receives the indication that a second has passed, it increments the seconds and, if the seconds were set back to zero, it increments the minutes. However, if the $AChronometer$ is asked to output its current $time$, it does so via channel out.

$$\begin{array}{l} Run \mathrel{\widehat=}\ tick \rightarrow IncSec;\ (sec = 0)\ \&\ IncMin \\ \qquad\qquad\qquad\qquad\qquad\quad \Box\ (sec \neq 0)\ \&\ Skip \\ \qquad\quad \Box\ time \rightarrow out!(min, sec) \rightarrow Skip \end{array}$$

The behaviour of process $AChronometer$ is represented by its main action: it initiates the

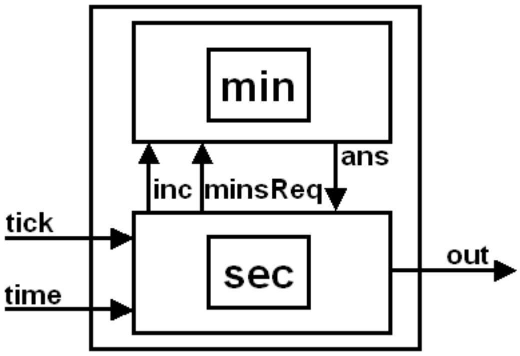


Figure 1.2: Concrete Chronometer

state components and then recursively executes its cycle. This concludes the specification of the abstract chronometer.

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

Figure 1.1 presents the external channels of the abstract chronometer.

In the development of the chronometer, we distribute the minutes and seconds in two different processes: *Seconds* and *Minutes*. The first one is responsible for the communication with the environment and for the *sec*onds; the second one is responsible for the *min*utes. Their interactions happen via three internal channels: *Seconds* indicates to *Minutes* that it must increment its number of minutes using channel *inc*. It may also request the current number of minutes via channel *misReq*; the answer is given via channel *ans*. The set of channels *Sync* groups these internal channels.

channel $inc, minsReq$
channel $ans : RANGE$

chanset $Sync \mathrel{\widehat=} \{inc, minsReq, ans\}$

The internal and external communications of the concrete chronometer are presented in Figure 1.2. The communication *minsReq* seems redundant; however, it removes a guarded output in the process *Minutes*, as we present later in this section. This is one of the restrictions of our translation to Java, as we discuss in Chapter 6; the concrete *Circus* specification cannot include output guards.

The refinement of the chronometer from the abstract to the concrete specification can be accomplished in only one refinement interaction. Since we do not intend to change the representation of minutes and seconds, this interaction involves no data refinement. In the action refinement, we change the *AChronometer* so that its state is composed of two partitions: one that contains the seconds (*Seconds*) and the other one that contains the minutes (*Minutes*). The intention is to split the actions of process *AChronometer* into two partitions: one is responsible for the interactions with the environment and the seconds, and the other one is responsible for the minutes. We name this refined process *Chronometer*.

$$\textbf{process } \mathit{Chronometer} \mathrel{\widehat=} \textbf{begin}$$

The *Seconds* state and the *Minutes* state are composed only of *sec* and *min*, respectively. The state of *Chronometer* is declared as the conjunction of these two schemes.

$$\begin{array}{l} \mathit{SecSt} \mathrel{\widehat=} [\, \mathit{sec} : \mathit{RANGE} \,] \\ \mathit{MinSt} \mathrel{\widehat=} [\, \mathit{min} : \mathit{RANGE} \,] \\ \textbf{state } \mathit{State} \mathrel{\widehat=} \mathit{SecSt} \land \mathit{MinSt} \end{array}$$

The first group of paragraphs access only *sec*, which is initialised to zero.

$$\mathit{SecInit} \mathrel{\widehat=} [\, \mathit{SecSt}' \mid \mathit{sec}' = 0 \,]$$

Another Z convention is used in the definitions that follow: for any *Schema*, $\Xi \mathit{Schema}$ represents the schema that includes both *Schema* and *Schema*$'$ and leaves the components values unchanged. The notation $\theta \mathit{Schema}$ denotes a binding (record) that group the values of the components of *Schema*.

$$\Xi \mathit{Schema} \mathrel{\widehat=} [\, \Delta \mathit{Schema} \mid \theta \mathit{Schema} = \theta \mathit{Schema}' \,]$$

Seconds are incremented by the operation *IncSec*.

$$\mathit{IncSec} \mathrel{\widehat=} [\, \Delta \mathit{SecSt};\ \Xi \mathit{MinSt} \mid \mathit{sec}' = (\mathit{sec} + 1) \mathbin{\mathsf{mod}} 60 \,]$$

The CSP action *RunSec* represents a cycle in the *Seconds* partition. If it receives the indication that a second has passed, it increments the seconds and, if the seconds were set back to zero, it sends a signal to the *Minutes* partition using channel *inc*. However, if it is asked to output the *time*, it asks the *Minutes* partition the number of minutes, receives the *ans*wer, and outputs the time via channel *out*.

$$\begin{array}{rl} \mathit{RunSec} \mathrel{\widehat=} & \mathit{tick} \to \mathit{IncSec};\ (\mathit{sec} = 0) \mathbin{\&} \mathit{inc} \to \mathit{Skip} \\ & \qquad\qquad\qquad\quad \Box\ (\mathit{sec} \neq 0) \mathbin{\&} \mathit{Skip} \\ & \Box\ \mathit{time} \to \mathit{minsReq} \to \mathit{ans}?\mathit{mins} \to \mathit{out}!(\mathit{mins}, \mathit{sec}) \to \mathit{Skip} \end{array}$$

The second group of paragraphs accesses only *min*, which is also initialised to zero.

$$\mathit{MinInit} \mathrel{\widehat=} [\, \mathit{MinSt}' \mid \mathit{min}' = 0 \,]$$

Minutes are incremented by the Z operation *IncMin*.

$$\mathit{IncMin} \mathrel{\widehat=} [\, \Delta \mathit{MinSt};\ \Xi \mathit{SecSt} \mid \mathit{min}' = (\mathit{min} + 1) \mathbin{\mathsf{mod}} 60 \,]$$

The CSP action *RunMin* represents a cycle in the *Minutes* partition. If it receives a request to increment the minutes, it does so. However, if the number of minutes is requested via channel *minsReq*, it outputs the value of *min* in channel *ans*.

$$\begin{array}{rl} \mathit{RunMin} \mathrel{\widehat=} & \mathit{inc} \to \mathit{IncMin} \\ & \Box\ \mathit{minsReq} \to \mathit{ans}!\mathit{min} \to \mathit{Skip} \end{array}$$

The main action of process *Chronometer* is the parallel composition of the behaviour of each partition: both initialise their state components and execute their cycles recursively. They synchronise on the channel set $Sync$, which is hidden from the external environment.

$$\bullet \left(\begin{array}{l} \left(\begin{array}{l} SecInit; \\ \quad (\mu X \bullet RunSec;\ X) \end{array} \right) \\ [\![\{sec\} \mid Sync \mid \{min\}]\!] \\ \left(\begin{array}{l} MinInit; \\ \quad (\mu X \bullet RunMin;\ X) \end{array} \right) \end{array} \right) \setminus Sync$$

end

The writing permissions are explicitly expressed in a *Circus* parallel composition of actions, as the one presented above. In this example, the *Seconds* partition may only modify the *sec* component and the *Minutes* partition may only modify *min*.

Proving the Refinement In order to prove that the concrete *Chronometer* is a refinement of the abstract one, we have to prove that its main action is indeed a refinement of the main action of process *AChronometer*.

$$\begin{array}{l} AInit;(\mu X \bullet Run;\ X) \\ \sqsubseteq_{\mathcal{A}} \\ ((SecInit;(\mu X \bullet RunSec;\ X)) \parallel (MinInit;(\mu X \bullet RunMin;\ X))) \setminus Sync \end{array}$$

For conciseness, in the remaining of this section, we abbreviate $[\![\{sec\} \mid Sync \mid \{min\}]\!]$ to $\parallel$, as we did above.

In *Circus*, action A_2 refines A_1 ($A_1 \sqsubseteq_{\mathcal{A}} A_2$), if, and only if, its behaviour never violates the behaviour of A_1. The action refinement laws reflects this relation; it is formally characterised using the UTP semantics of *Circus*. The process of refining actions consists of repeatedly applying these laws until we reach the desired concrete action.

As an example, we have the law that splits an initialisation operation into a sequence. The side conditions of some of the refinement laws involve meta-functions such as α, FV, DFV, and $UDFV$. The function α determines the set of components of a given schema; FV is a function that defines the free variables of a predicate or expression; finally, for a given predicate p, the functions DFV and $UDFV$ yield the dashed and the undashed free variables of p, respectively.

Law C.72 (Initialisation schema/Sequence—introduction*).

$$[\, S_1';\ S_2' \mid CS_1 \land CS_2 \,] = [\, S_1' \mid CS_1 \,];[\, S_2' \mid CS_2 \,]$$

provided

- ➭ $\alpha(S_1) \cap \alpha(S_2) = \varnothing$
- ➭ $DFV(CS_1) \subseteq \alpha(S_1')$
- ➭ $DFV(CS_2) \subseteq \alpha(S_2')$ ❑

This laws applies to a schema S which operates over a state composed of two disjoint partitions S_1' and S_2'. The updates of S on the state are expressed as a conjunction of two predicates CS_1 and CS_2, whose free-variables are in the disjoint sets of components of S_1' and S_2', respectively. It transforms the given schema into a sequence of two schemas; each of them corresponds to the original operation on one of the state partitions.

We start our proof from the main action of the abstract process.

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

The abstract initialisation of the state meets all the provisos of the Law C.72. For this reason we may split it into a sequence of two different initialisations: the initialisation of the seconds and the initialisation of the minutes.

$$\begin{array}{l} = [C.72] \\ MinInit;\ SecInit; (\mu X \bullet Run;\ X) \end{array}$$

Throughout this work, we use the notation $A_1 \sqsubseteq_{\mathcal{A}} [\,law_1, \ldots, law_n\,]\{po_1\}\ldots\{po_n\}\ A_2$ to denote that A_1 may be refined to A_2 using laws $law_1, \ldots, law_n$, if the proof obligations $po_1, \ldots, po_n$ hold. For conciseness, in this introduction we omit the proof obligations and informally justify the validity of the law applications.

In the next step, we use the least fixed-point law to split the single recursion into the parallel composition of two recursions: one is concerned with the behaviour of the *Seconds* and the other is concerned with the behaviour of the *Minutes*. This is justified by Lemma 1.1 proved latter in this section.

$$\begin{array}{l} \sqsubseteq_{\mathcal{A}} [C.129] \\ MinInit;\ SecInit; ((\mu X \bullet RunSec;\ X) \parallel (\mu X \bullet RunMin;\ X)) \setminus Sync \end{array}$$

Next, since schema expressions use no channels, we may expand the hiding.

$$\begin{array}{l} = [C.120, C.125] \\ (MinInit;\ SecInit; (\mu X \bullet RunSec;\ X) \parallel (\mu X \bullet RunMin;\ X)) \setminus Sync \end{array}$$

Finally, the schemas change only variables declared in one of the partitions of the parallel composition, and the variables they write to are not used by the other side of the parallel composition. For this reason, we may move each of them to one of the sides of the composition.

$$\begin{array}{l} = [C.73] \\ ((SecInit; (\mu X \bullet RunSec;\ X)) \parallel (MinInit;\ (\mu X \bullet RunMin;\ X))) \setminus Sync \end{array}$$

This concludes the proof of the action refinement. However, we are still left with the proof of the following lemma.

Lemma 1.1

$$\begin{array}{l} ((\mu X \bullet RunSec;\ X) \parallel (\mu X \bullet RunMin;\ X)) \setminus Sync \\ = Run;\ ((\mu X \bullet RunSec;\ X) \parallel (\mu X \bullet RunMin;\ X)) \setminus Sync \end{array}$$

Starting from the left-hand side of the lemma, we unfold the first recursion. Afterwards, we distribute the recursion through each of the external choices that are in the recursion body. Then, we combine the second recursive program in parallel with each of the branches of the first recursion. The strategy is then to show that each of these branches can be transformed into a branch of *Run* followed by the left-hand side itself. This results in a program which coincides with the body of the recursion on the right-hand side of the lemma, except that in place of the recursive call we have the left-hand side itself. The distribution laws and the definition of action *Run* concludes this proof. In what follows, we present the details of this proof.

As previously explained, we start our proof by unfolding the recursion in the left-hand side of the parallel composition; furthermore, we apply the definition of *RunSec*.

$$
\begin{array}{l}
((\mu X \bullet \mathit{RunSec};\ X) \parallel (\mu X \bullet \mathit{RunMin};\ X)) \setminus \mathit{Sync} \\
= [C.128, \text{Definition of } \mathit{RunSec}] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{tick} \rightarrow \mathit{IncSec};\ (\mathit{sec} = 0)\ \&\ \mathit{inc} \rightarrow \mathit{Skip} \\
\qquad\qquad\qquad \Box\ (\mathit{sec} \neq 0)\ \&\ \mathit{Skip} \\
\Box\ \mathit{time} \rightarrow \mathit{minsReq} \rightarrow \mathit{ans}?\mathit{mins} \rightarrow \\
\qquad \mathit{out}!(\mathit{mins}, \mathit{sec}) \rightarrow \mathit{Skip}
\end{array}\right);\ (\mu X \bullet \mathit{RunSec};\ X) \\
\parallel \\
(\mu X \bullet \mathit{RunMin};\ X)
\end{array}\right) \setminus \mathit{Sync}
\end{array}
$$

Afterwards, we distribute the recursion through each of the external choices. For this purpose, we use the following law:

Law C.112 (External choice/Sequence—distribution).

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

It distributes an action through an external choice of guarded actions. In our example, by applying this law, we get the following result.

$$
\begin{array}{l}
= [C.112] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathit{tick} \rightarrow \mathit{IncSec}; \\
\qquad \left(\begin{array}{l}
(\mathit{sec} = 0)\ \&\ \mathit{inc} \rightarrow \mathit{Skip} \\
\Box\ (\mathit{sec} \neq 0)\ \&\ \mathit{Skip}
\end{array}\right);\ (\mu X \bullet \mathit{RunSec};\ X) \\
\\
\Box\ \mathit{time} \rightarrow \mathit{minsReq} \rightarrow \mathit{ans}?\mathit{mins} \rightarrow \\
\qquad \mathit{out}!(\mathit{mins}, \mathit{sec}) \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunSec};\ X)
\end{array}\right) \\
\parallel \\
(\mu X \bullet \mathit{RunMin};\ X)
\end{array}\right) \setminus \mathit{Sync}
\end{array}
$$

The next distribution law states that the distribution is also valid if the guards are mutually exclusive. In this case, we do not need to assure that there are any communications happening: A_1 and A_2 may be any *Circus* action, not necessarily a communication.

Law C.113 (External choice/Sequence—distribution 2*).

$$((g_1 \,\&\, A_1) \,\Box\, (g_2 \,\&\, A_2));\ B = ((g_1 \,\&\, A_1);\ B) \,\Box\, ((g_2 \,\&\, A_2);\ B)$$

provided $g_1 \Rightarrow \neg\, g_2$ ❑

In our example, the distribution through the choices of the *tick* branch is valid since the guards are indeed mutually exclusive.

$$= [C.113]$$

$$\left(\begin{array}{l} \left(\begin{array}{l} tick \to IncSec;\ (sec = 0) \,\&\, inc \to Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \qquad\qquad\qquad\qquad \Box\, (sec \neq 0) \,\&\, Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \Box\, time \to minsReq \to ans?mins \to \\ \qquad out!(mins, sec) \to Skip;\ (\mu\, X \bullet RunSec;\ X) \end{array} \right) \\ \| \\ (\mu\, X \bullet RunMin;\ X) \end{array} \right) \setminus Sync$$

From the definition of *RunMin*, it is trivial that this action is firstly willing to synchronise. For this reason, we may distribute the parallel composition over the external choice of the left-hand side of the parallel composition as follows.

$$= [C.87]$$

$$\left(\begin{array}{l} \left(\begin{array}{l} tick \to IncSec;\ (sec = 0) \,\&\, inc \to Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \qquad\qquad\qquad\qquad \Box\, (sec \neq 0) \,\&\, Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \| \\ (\mu\, X \bullet RunMin;\ X) \end{array} \right) \\ \Box \left(\begin{array}{l} time \to minsReq \to ans?mins \to \\ \qquad out!(mins, sec) \to Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \| \\ (\mu\, X \bullet RunMin;\ X) \end{array} \right) \end{array} \right) \setminus Sync$$

In order to apply the step law below, besides guaranteeing that *RunMin* is firstly willing to synchronise, which we have already done, we also have to guarantee that the events *tick* and *time* may happen independently. This condition is also met because neither of these events are in the synchronisation channel set.

$$= [C.100, C.84]$$

$$\left(\begin{array}{l} tick \to IncSec; \\ \qquad \left(\begin{array}{l} \left(\begin{array}{l} (sec = 0) \,\&\, inc \to Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \Box\, (sec \neq 0) \,\&\, Skip;\ (\mu\, X \bullet RunSec;\ X) \end{array} \right) \\ \| \\ (\mu\, X \bullet RunMin;\ X) \end{array} \right) \\ \Box\, time \to Skip; \\ \qquad \left(\begin{array}{l} minsReq \to ans?mins \to \\ \qquad out!(mins, sec) \to Skip;\ (\mu\, X \bullet RunSec;\ X) \\ \| \\ (\mu\, X \bullet RunMin;\ X) \end{array} \right) \end{array} \right) \setminus Sync$$

Next, the distribution of the hiding over the external choice is valid because the initial events in the choice are not being hidden.

$$
\begin{array}{l}
= [C.122, C.120, C.125] \\
\left.\begin{array}{l}
\left.\begin{array}{l}
\quad \mathit{tick} \rightarrow \mathit{IncSec}; \\
\qquad \left(\begin{array}{l}
\left(\begin{array}{l}
(\mathit{sec} = 0) \mathbin{\&} \mathit{inc} \rightarrow \mathit{Skip}; (\mu X \bullet \mathit{RunSec}; X) \\
\Box\ (\mathit{sec} \neq 0) \mathbin{\&} \mathit{Skip}; (\mu X \bullet \mathit{RunSec}; X)
\end{array}\right) \\
\parallel \\
(\mu X \bullet \mathit{RunMin}; X)
\end{array}\right) \setminus \mathit{Sync}
\end{array}\right\} (1) \\
\left.\begin{array}{l}
\quad \Box\ \mathit{time} \rightarrow \mathit{Skip}; \\
\qquad \left(\begin{array}{l}
\mathit{minsReq} \rightarrow \mathit{ans}?\mathit{mins} \rightarrow \mathit{out}!(\mathit{mins}, \mathit{sec}) \rightarrow \mathit{Skip}; \\
\qquad (\mu X \bullet \mathit{RunSec}; X) \\
\parallel \\
(\mu X \bullet \mathit{RunMin}; X)
\end{array}\right) \setminus \mathit{Sync}
\end{array}\right\} (2)
\end{array}\right\} (3)
\end{array}
$$

The strategy now is to show that each of these choices can be transformed into the choice in the right-hand side of this lemma, followed by the left-hand side itself. We start with the first branch: once again, since the action *RunMin* is firstly willing to synchronise, we may distribute the parallel composition over the choice in the first branch.

$$
\begin{array}{l}
(1) \\
= [C.87] \\
\mathit{tick} \rightarrow \mathit{IncSec}; \left(\begin{array}{l}
\left(\begin{array}{l}
(\mathit{sec} = 0) \mathbin{\&} \mathit{inc} \rightarrow \mathit{Skip}; (\mu X \bullet \mathit{RunSec}; X) \\
\parallel \\
(\mu X \bullet \mathit{RunMin}; X)
\end{array}\right) \\
\Box \left(\begin{array}{l}
(\mathit{sec} \neq 0) \mathbin{\&} \mathit{Skip}; (\mu X \bullet \mathit{RunSec}; X) \\
\parallel \\
(\mu X \bullet \mathit{RunMin}; X)
\end{array}\right)
\end{array}\right) \setminus \mathit{Sync}
\end{array}
$$

The associativity of guard and sequence justifies the next step of our refinement, as follows.

$$
\begin{array}{l}
= [C.132, C.59] \\
\mathit{tick} \rightarrow \mathit{IncSec}; \left(\begin{array}{l}
\left(\begin{array}{l}
((\mathit{sec} = 0) \mathbin{\&} \mathit{Skip}); \mathit{inc} \rightarrow \mathit{Skip}; \\
\qquad (\mu X \bullet \mathit{RunSec}; X) \\
\parallel \\
(\mu X \bullet \mathit{RunMin}; X)
\end{array}\right) \\
\Box \left(\begin{array}{l}
((\mathit{sec} \neq 0) \mathbin{\&} \mathit{Skip}); (\mu X \bullet \mathit{RunSec}; X) \\
\parallel \\
(\mu X \bullet \mathit{RunMin}; X)
\end{array}\right)
\end{array}\right) \setminus \mathit{Sync}
\end{array}
$$

Once again, we may apply the step law in order to move the guards out of the parallel composition. This is valid since the action *RunMin* is firstly willing to synchronise and the actions $(sec = 0)$ & *Skip* and $(sec \neq 0)$ & *Skip* have no communications.

$$= [C.84, C.59, C.132]$$
$$tick \rightarrow IncSec; \left(\begin{array}{l} (sec = 0)\ \&\ \left(\begin{array}{l} (inc \rightarrow Skip;\ (\mu X \bullet RunSec;\ X)) \\ \parallel \\ (\mu X \bullet RunMin;\ X) \end{array} \right) \\ \Box\ (sec \neq 0)\ \&\ Skip;\ \left(\begin{array}{l} (\mu X \bullet RunSec;\ X) \\ \parallel \\ (\mu X \bullet RunMin;\ X) \end{array} \right) \end{array} \right) \setminus Sync$$

Since the guards are mutually exclusive, we may distribute the hiding over the external choice. Furthermore, they can also be distributed over the guards.

$$= [C.123]$$
$$tick \rightarrow IncSec; \left(\begin{array}{l} (sec = 0)\ \& \\ \qquad \left. \left(\begin{array}{l} (inc \rightarrow Skip;\ (\mu X \bullet RunSec;\ X)) \\ \parallel \\ (\mu X \bullet RunMin;\ X) \end{array} \right) \setminus Sync \right\} (4) \\ \Box\ (sec \neq 0)\ \& \\ \qquad \left. Skip;\ \left(\begin{array}{l} (\mu X \bullet RunSec;\ X) \\ \parallel \\ (\mu X \bullet RunMin;\ X) \end{array} \right) \setminus Sync \right\} (5) \end{array} \right)$$

The second branch (5) is already in the desired format. We turn our attention to the first branch. First, we unfold the recursion in the right-hand side of the parallel composition and apply the definition of *RunMin*.

$$(4)$$
$$= [C.128, \text{Definition of } RunMin]$$
$$\left(\begin{array}{l} (inc \rightarrow Skip;\ (\mu X \bullet RunSec;\ X)) \\ \parallel \\ \left(\begin{array}{l} inc \rightarrow IncMin \\ \Box\ minsReq \rightarrow ans!min \rightarrow Skip \end{array} \right);\ (\mu X \bullet RunMin;\ X) \end{array} \right) \setminus Sync$$

Since *minsReq* is in the synchronisation channel, the second choice never actually happens; it may, therefore, be removed.

$$= [C.86]$$
$$((inc \rightarrow Skip;\ (\mu X \bullet RunSec;\ X)) \parallel (inc \rightarrow IncMin;\ (\mu X \bullet RunMin;\ X))) \setminus Sync$$

Next, since the communication *inc* is being hidden, it may also be removed.

$$= [C.83]$$
$$((Skip;\ (\mu X \bullet RunSec;\ X)) \parallel (IncMin;\ (\mu X \bullet RunMin;\ X))) \setminus Sync$$

Finally, the schema *IncMin* changes only variables declared in the *Minutes* partition, and they are not used by the *Seconds* partition. For this reason, we may move *IncMin* away from the parallel composition.

$$
\begin{array}{l}
= [C.73] \\
\mathit{IncMin};\ ((\mu X \bullet \mathit{RunSec};\ X) \parallel (\mu X \bullet \mathit{RunMin};\ X)) \setminus \mathit{Sync}
\end{array}
$$

This concludes the refinement of the first branch (1). We turn our attention back to the second branch. We also start this refinement by unfolding the recursion in the right-hand side of the parallel composition and by applying the definition of *RunMin*.

$$
\begin{array}{l}
(2) \\
= [C.128, \text{Definition of } \mathit{RunMin}] \\
\mathit{time} \rightarrow \mathit{Skip}; \\
\quad \left(\begin{array}{l}
\mathit{minsReq} \rightarrow \mathit{ans}?\mathit{mins} \rightarrow \\
\qquad \mathit{out}!(\mathit{mins}, \mathit{sec}) \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunSec};\ X) \\
\parallel \\
\left(\begin{array}{l}
\mathit{inc} \rightarrow \mathit{IncMin} \\
\Box\ \mathit{minsReq} \rightarrow \mathit{ans}!\mathit{min} \rightarrow \mathit{Skip}
\end{array} \right);\ (\mu X \bullet \mathit{RunMin};\ X)
\end{array} \right) \setminus \mathit{Sync}
\end{array}
$$

Since *inc* is in the synchronisation channel, the first choice never happens; it may, therefore, be removed.

$$
\begin{array}{l}
= [C.86] \\
\mathit{time} \rightarrow \mathit{Skip}; \\
\quad \left(\begin{array}{l}
\mathit{minsReq} \rightarrow \mathit{ans}?\mathit{mins} \rightarrow \\
\qquad \mathit{out}!(\mathit{mins}, \mathit{sec}) \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunSec};\ X) \\
\parallel \\
\mathit{minsReq} \rightarrow \mathit{ans}!\mathit{min} \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunMin};\ X)
\end{array} \right) \setminus \mathit{Sync}
\end{array}
$$

The synchronisation in *minsReq* may be removed because it is being hidden.

$$
\begin{array}{l}
= [C.83] \\
\mathit{time} \rightarrow \mathit{Skip}; \\
\quad \left(\begin{array}{l}
\mathit{ans}?\mathit{minsReq} \rightarrow \\
\qquad \mathit{out}!(\mathit{minsReq}, \mathit{sec}) \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunSec};\ X) \\
\parallel \\
\mathit{ans}!\mathit{min} \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunMin};\ X)
\end{array} \right) \setminus \mathit{Sync}
\end{array}
$$

For the same reason, we may also remove the communication *ans*; however, the value communicated must be used by the left-hand side of the parallel composition.

$$
\begin{array}{l}
= [C.81, C.132] \\
\mathit{time} \rightarrow \mathit{Skip}; \\
\quad ((\mathit{out}!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip};\ (\mu X \bullet \mathit{RunSec};\ X)) \parallel (\mu X \bullet \mathit{RunMin};\ X)) \setminus \mathit{Sync}
\end{array}
$$

The event *time* is not in the synchronisation channel set and the action *RunMin* is firstly willing to synchronise; once again, we may apply the step law.

$$
\begin{array}{l}
= [C.128, C.84, C.132] \\
\mathit{time} \rightarrow \mathit{Skip}; \\
\quad (\mathit{out}!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}; ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X))) \setminus \mathit{Sync}
\end{array}
$$

Finally, since the events *time* and *out* are not being hidden, we may move the hiding as follows.

$$
\begin{array}{l}
= [C.120, C.125, C.100] \\
\mathit{time} \rightarrow \mathit{out}!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}; ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X)) \setminus \mathit{Sync}
\end{array}
$$

This concludes the refinement of this branch. We return to the proof of the main lemma.

$$
\begin{array}{l}
(3) \\
= \\
\mathit{tick} \rightarrow \mathit{IncSec}; \\
\quad (\mathit{sec} = 0) \mathrel{\&} \mathit{IncMin}; ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X)) \setminus \mathit{Sync} \\
\quad \Box\ (\mathit{sec} \neq 0) \mathrel{\&} \mathit{Skip}; ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X)) \setminus \mathit{Sync} \\
\Box\ \mathit{time} \rightarrow \mathit{out}!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}; \\
\quad ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X)) \setminus \mathit{Sync}
\end{array}
$$

However, as in the early stages of this proof, this is the result of distributing the entire recursion through each of the choices inside the recursion body.

$$
\begin{array}{l}
= [C.112, C.113] \\
\left(\begin{array}{l}
\mathit{tick} \rightarrow \mathit{IncSec}; \\
\qquad (\mathit{sec} = 0) \mathrel{\&} \mathit{IncMin} \\
\qquad \Box\ (\mathit{sec} \neq 0) \mathrel{\&} \mathit{Skip} \\
\Box\ \mathit{time} \rightarrow \mathit{out}!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}
\end{array}\right); \\
\quad ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X)) \setminus \mathit{Sync}
\end{array}
$$

The definition of *Run* concludes this proof.

$$
\begin{array}{l}
= [\text{Definition of } \mathit{Run}] \\
\mathit{Run}; ((\mu X \bullet \mathit{RunSec}; X) \parallel (\mu X \bullet \mathit{RunMin}; X)) \setminus \mathit{Sync}
\end{array}
$$

❑

After this action refinement, we have a process with a state partitioned into two: one is concerned with the seconds and the other one is concerned with the minutes. Each partition has its own set of paragraphs, which are disjoint, since no action in one changes a state component in the other. The main action of the refined process is defined in terms of the parallel composition of actions from both partitions. The final step of our refinement uses the process refinement Law C.146 in order to rewrite the process *Chronometer* in terms of two independent processes as follows.

$$
\begin{array}{l}
\textbf{process } \mathit{Seconds} \mathrel{\widehat=} \textbf{ begin state } \mathit{SecSt} \mathrel{\widehat=} [\, \mathit{sec} : \mathit{RANGE} \,] \\
\quad \mathit{SecInit} \mathrel{\widehat=} [\, \mathit{SecSt}' \mid \mathit{sec}' = 0 \,] \\
\quad \mathit{IncSec} \mathrel{\widehat=} [\, \Delta \mathit{SecSt} \mid \mathit{sec}' = (\mathit{sec} + 1) \mathsf{\ mod\ } 60 \,] \\
\quad \mathit{RunSec} \mathrel{\widehat=} \mathit{tick} \rightarrow \mathit{IncSec};\ (\mathit{sec} = 0)\ \&\ \mathit{inc} \rightarrow \mathit{Skip} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Box\ (\mathit{sec} \neq 0)\ \&\ \mathit{Skip} \\
\qquad\qquad\qquad \Box\ \mathit{time} \rightarrow \mathit{minsReq} \rightarrow \mathit{ans}?\mathit{mins} \rightarrow \mathit{out}!(\mathit{mins}, \mathit{sec}) \rightarrow \mathit{Skip} \\
\quad \bullet\ \mathit{SecInit};(\mu X \bullet \mathit{RunSec};\ X) \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textbf{process } \mathit{Minutes} \mathrel{\widehat=} \textbf{ begin state } \mathit{MinSt} \mathrel{\widehat=} [\, \mathit{min} : \mathit{RANGE} \,] \\
\quad \mathit{MinInit} \mathrel{\widehat=} [\, \mathit{MinSt}' \mid \mathit{min}' = 0 \,] \\
\quad \mathit{IncMin} \mathrel{\widehat=} [\, \Delta \mathit{MinSt} \mid \mathit{min}' = (\mathit{min} + 1) \mathsf{\ mod\ } 60 \,] \\
\quad \mathit{RunMin} \mathrel{\widehat=} \mathit{inc} \rightarrow \mathit{IncMin} \\
\qquad\qquad\qquad \Box\ \mathit{minsReq} \rightarrow \mathit{ans}!\mathit{min} \rightarrow \mathit{Skip} \\
\quad \bullet\ \mathit{MinInit};(\mu X \bullet \mathit{RunMin};\ X) \\
\textbf{end}
\end{array}
$$

$$
\textbf{process } \mathit{Chronometer} \mathrel{\widehat=} (\mathit{Seconds} \lbrack\!\lbrack \mathit{Sync} \rbrack\!\rbrack \mathit{Minutes}) \setminus \mathit{Sync}
$$

Using the Z refinement calculus [29], we may further refine the processes *Seconds* and *Minutes*, transforming the schema operations into single assignments. For instance, the refinement of the body of the action *SecInit* would result in the assignment $sec := 0$. After this trivial refinement, we end with an implementable *Circus* specification.

The application of the translation strategy presented in Chapter 6 to this specification results in the Java code that implements the Chronometer, which can be found in [71]. Besides some auxiliary classes, which are explained in Chapter 6, the Java code contains three classes `Seconds`, `Minutes`, and `Chronometer`; they implement the behaviour of the processes discussed in this section.

Figure 1.3 summarises the application of our method to this example. Although simple, this example illustrates our approach by deriving an implementation of a chronometer from its abstract specification. The development of a larger scale case study is the topic of Chapter 5, where we develop a fire control system.

1.4 Outline

In Chapter 2, we present an introduction to *Circus* and the UTP. Using a simple example of a *Register*, we describe the *Circus* constructors for describing processes and their constituent actions.

We start Chapter 3 by presenting the denotational semantics of *Circus*. Next in this chapter, we present our steps towards a theorem prover for *Circus*. We present the mechanisation of the theories presented in the UTP: the theories of relation, designs, reactive and CSP processes. These theories are the basis of the *Circus* theory, whose presentation concludes this chapter.

Chapter 4 discusses the refinement notions for *Circus* processes and their constituent actions. The simulation technique and the refinement strategy presented in [27] are also

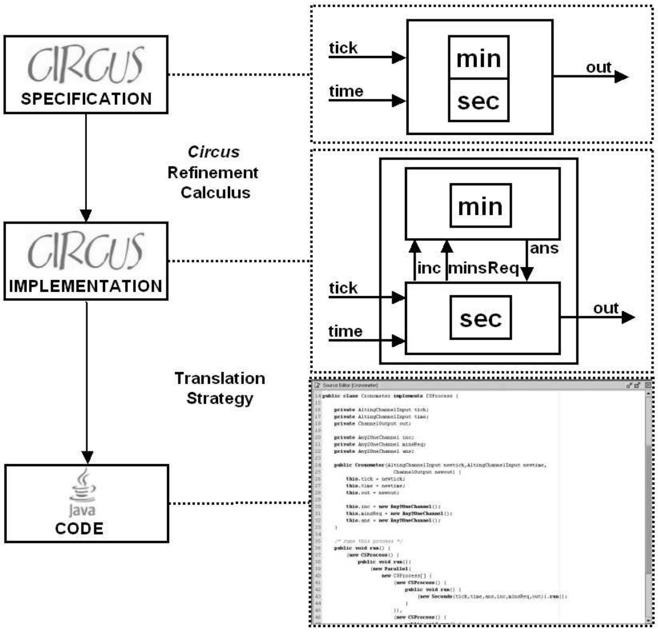


Figure 1.3: Summary of the Chronometer Example

discussed in this chapter. Next, this chapter presents some of the new refinement laws proposed in this work and the corrections made to some of the previously proposed *Circus* refinement laws. We conclude this chapter with a discussion of the soundness proofs of some of the refinement laws.

Chapter 5 presents a safety-critical fire control system: a case study on the refinement calculus of *Circus*. First, we informally describe the system and present its abstract centralised specification. Then, we describe the refinement strategy adopted in the development of a distributed concrete specification of the fire control system.

In Chapter 6, a strategy for implementing *Circus* programs in Java is presented. First, we present a brief introduction to JCSP [99, 98], a Java library that can be used to support the implementation of CSP programs in Java. Then, we present the translation strategy itself in a didactic account. The strategy is presented for a large subset of executable *Circus*. Then, we extend the strategy in order to deal with synchronisation channels and the *Circus* indexing operator, described in Chapter 2. Next, we describe the translation of generic and multi-synchronised channels. Finally, we describe how we have applied this translation strategy to obtain an implementation in Java of our case study presented in Chapter 5.

Chapter 7 concludes this document. It discusses how the results that we present support our thesis, and gives an account of related and future work.

Chapter 2

Background

This chapter introduces the background of this thesis. Section 2.1 presents *Circus* and discuss its operators in more detail. A simple example, a *Register* is used to illustrate these operators. Finally, in Section 2.2, we describe the theoretical basis of *Circus*, the *Unifying Theories of Programming.*

2.1 *Circus*

Circus is a language that is suitable for the specification of concurrent and reactive systems; it also has a theory of refinement associated to it. Its objective is to give a sound basis for the development of concurrent and distributed system in a calculational style like that of [63]. In the sections that follows, we introduce *Circus* based on its syntax, which can be found in Appendix A.

2.1.1 *Circus* Programs

Circus is based on imperative CSP [80], and adds specification facilities in the Z [107] style. This enables both state and communications aspects to be captured in the same specification, as in [90]. In the same way as Z specifications, *Circus* programs are formed by a sequence of paragraphs.

$$\mathsf{Program} \quad ::= \quad \mathsf{CircusPar}^*$$

Here, $\mathsf{CircusPar}^*$ denotes a possibly empty list of elements of the syntactic category $\mathsf{CircusPar}$ of *Circus* paragraphs.

Each of these paragraphs can either be a Z paragraph [91], here denoted by the syntactic category Par, a definition of channels, a channel set definition, or a process declaration.

$$\mathsf{CircusPar} \quad ::= \quad \mathsf{Par} \mid \mathbf{channel}\ \mathsf{CDecl} \mid \mathbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \mid \mathsf{ProcDecl}$$

The syntactic category N is that of valid Z identifiers.

We illustrate the main constructs of *Circus* using the specification of a simple register (Figure 2.1). It is initialised with zero, and can store or add a given value to its current value. It can also output or reset its current value. The specification is composed of seven paragraphs.

2.1.2 Channel Declarations

All the channels that are used within a process must be declared. The syntactic categories Exp and $\mathsf{SchemaExp}$ are those of Z expressions and schema expressions defined in [91]. Here, N^+ denotes a non-empty list of elements of the syntactic category N.

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

In a channel declaration, we declare the name of the channel and the type of the values it can communicate. However, if the channel does not communicate any value, but it is used only as a synchronising event, its declaration contains only its name; no type is defined. A channel declaration may declare more than one channel of the same type. In this case, instead of a single channel name, we have a comma-separated list of channel names. This is illustrated in Figure 2.1 by the declaration of channels *store*, *add*, and *out*.

Generic channel declarations introduce a family of channels. For instance, the declaration **channel** $[T]\ c : T$ declares a family of channels c. For every actual type S, we have a channel $c[S]$ that communicates values of type S. Channels can also be declared using schemas that group channel declarations, but do not have a predicate part. This follows from the fact that the only restriction that may be imposed on a channel is the type it communicates.

2.1.3 Channel Set Declarations

We may introduce sets of previously defined channels in a **chanset** paragraph. In this case, we declare the name of the set and a channel-set expression, which determines the channels that are members of this set. The empty set of channels $\{\!|\,|\!\}$, channel enumerations enclosed in $\{\!|$ and $|\!\}$, and expressions formed by some of the Z set operators are the elements of the syntactic category CSExp. In our example, we declare the alphabet of the *Register* as the channel set *RegAlphabet*. These are the channels that can be used to interact with this process.

2.1.4 Process Declarations

The declaration of a process is composed of its name and its definition. Furthermore, like channels, processes may also be declared generic. In this case, the declaration introduces a family of processes.

$$\mathsf{ProcDecl} \ ::= \ \textbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{ProcDef} \mid \textbf{process}\ \mathsf{N}[\mathsf{N}^+] \mathrel{\widehat=} \mathsf{ProcDef}$$

A process is specified as a (possibly) parametrised process, or as an indexed process. If a process is parametrised or indexed, we first have the declaration of its parameters.

$$
\begin{array}{l}
\textbf{channel } store, add, out : \mathbb{Z} \\
\textbf{channel } result, reset \\
\textbf{process } Register \mathrel{\widehat=} \\
\quad \textbf{begin state } RegSt \mathrel{\widehat=} [value : \mathbb{Z}] \\
\quad RegCycle \mathrel{\widehat=} \begin{array}[t]{l} store?newValue \rightarrow value := newValue \\ \Box\ add?newValue \rightarrow value := value + newValue \\ \Box\ result \rightarrow out!value \rightarrow Skip \\ \Box\ reset \rightarrow value := 0 \end{array} \\
\quad \bullet\ value := 0;\ (\mu X \bullet RegCycle;\ X) \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textbf{channel } read, write : \mathbb{Z} \\
\textbf{process } SumClient \mathrel{\widehat=} \\
\quad \textbf{begin} \\
\quad ReadValue \mathrel{\widehat=} read?n \rightarrow reset \rightarrow Sum(n) \\
\quad Sum \mathrel{\widehat=}\ n : \mathbb{Z} \bullet \begin{array}[t]{l} (n = 0)\ \&\ result \rightarrow out?r \rightarrow write!r \rightarrow Skip \\ \Box\ (n \neq 0)\ \&\ add!n \rightarrow Sum(n-1) \end{array} \\
\quad \bullet\ \mu X \bullet ReadValue;\ X \\
\quad \textbf{end} \\
\textbf{chanset } RegAlphabet == \{\!|\ store, add, out, result, reset\ |\!\} \\
\textbf{process } Summation \mathrel{\widehat=} (SumClient \mathrel{[\![} RegAlphabet \mathrel{]\!]} Register) \setminus RegAlphabet
\end{array}
$$

Figure 2.1: A Simple Register

The syntactic category Decl is the same as in [91]. Afterwards, following a $\bullet$, in the case of parametrised processes, or a $\odot$, in the case of indexed processes, we have the declaration of the process body. In both cases, the parameters may be used as local variables in the definition of the process. If the process is not parametrised, we have only the definition of its body.

$$\mathsf{ProcDef} \quad ::= \quad \mathsf{Decl} \bullet \mathsf{ProcDef} \mid \mathsf{Decl} \odot \mathsf{ProcDef} \mid \mathsf{Proc}$$

A process may be explicitly defined, or it may be defined in terms of other processes (compound processes). An explicit process definition is delimited by the keywords **begin** and **end**; it is formed by a sequence of process paragraphs and a distinguished nameless main action, which defines the process behaviour, in the end. Furthermore, in *Circus* we use the Z notation to define the state of a process. It is described as a schema paragraph, after the keyword **state**.

$$
\begin{array}{lll}
\mathsf{Proc} & ::= & \textbf{begin } \mathsf{PPar}^* \textbf{ state } \mathsf{SchemaExp}\ \mathsf{PPar}^* \bullet \mathsf{Action} \textbf{ end} \\
 & | & \ldots
\end{array}
$$

Process *Register* in Figure 2.1 is defined in this way. The schema *RegState* describes the internal state of the process *Reg*: it contains an integer *value* that stores its value.

The behaviour of *Register* is described by the unnamed action after a $\bullet$. The process *Register* has a recursive behaviour: after its initialisation, it behaves like *RegCycle*, and then recurses.

2.1.5 Compound Processes

Processes may also be defined in terms of other previously defined processes using the process name, CSP operators, iterated CSP operators, or indexed operators, which are particular to *Circus* specifications.

$$
\begin{array}{lll}
\mathsf{Proc} & ::= & \ldots \\
& | & \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \mathbin{\Box} \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
& | & \mathsf{Proc} \mathbin{[\![} \mathsf{CSExp} \mathbin{]\!]} \mathsf{Proc} \mid \mathsf{Proc} \mathbin{|||} \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
& | & (\mathsf{Decl} \bullet \mathsf{ProcDef})(\mathsf{Exp}^{+}) \mid \mathsf{N}(\mathsf{Exp}^{+}) \mid \mathsf{N} \\
& | & (\mathsf{Decl} \odot \mathsf{ProcDef})\lfloor \mathsf{Exp}^{+} \rfloor \mid \mathsf{N}\lfloor \mathsf{Exp}^{+} \rfloor \mid \mathsf{Proc}[\mathsf{N}^{+} := \mathsf{N}^{+}] \mid \mathsf{N}[\mathsf{Exp}^{+}] \\
& | & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \Box\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \sqcap\ \mathsf{Decl} \bullet \mathsf{Proc} \\
& | & [\![\mathsf{CSExp}]\!]\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ |||\ \mathsf{Decl} \bullet \mathsf{Proc}
\end{array}
$$

Processes P_1 and P_2 can be combined in sequence using the sequence operator: $P_1;P_2$. This process executes the process P_2 after the execution of P_1 terminates. The external choice $P_1 \mathbin{\Box} P_2$ initially offers events of both processes. The performance of the first event resolves the choice in favour of the process that performs it. Differently from the external choice, the environment has no control over the internal choice $P_1 \sqcap P_2$, in which the process internally (nondeterministically) resolves the choice.

The parallel operator follows the alphabetised parallel operator approach adopted by [80]; we must declare a synchronisation channel set. For instance, in $P_1 \mathbin{[\![} cs \mathbin{]\!]} P_2$ the processes P_1 and P_2 synchronise on the channels in the set cs; events that are not listed occur independently. By way of illustration, the process *Summation* in Figure 2.1 reads a value n through channel *read*, interacts with a *Register*, and outputs the value of $\sum_{i=1}^{n} i$ through channel *write*. It is declared as a parallel composition of processes *Register* and its client *SumClient*; they synchronise on the set of events *RegAlphabet*.

Processes can also be composed in interleaving. For instance, a process *RegisterTwice* that represents two *Register*s running independently can be defined as the composition *Register* $|||$ *Register*. However, an event *reset* leads to a non-deterministic choice of which *Register* process of the interleaving actually starts: one of the processes resets, and the other one does not.

The event hiding operator $P \setminus cs$ is used to encapsulate the events that are in the channel set cs. This removes these events from the interface of P, which become no longer visible to the environment. For instance, the process *Summation* encapsulates the interaction between the processes *Register* and *SumClient* (*RegAlphabet*); the only ways to interact with *Summation* are via the channels *write* and *read*.

As with CSP, *Circus* provides finite iterated operators that can be used to generalise the binary operators of sequence, external and internal choice, parallel composition, and interleaving. Furthermore, we may instantiate a parametrised process by providing values for each of its parameters. For instance, we may have either $P(v)$, where $P \mathrel{\widehat=} (x : T \bullet Proc)$, or, for reasoning purposes, we can write directly $(x : T \bullet Proc)(v)$. Apart from sequence, all the iterated operators are commutative and associative. For this reason, there is no concern about the order of the elements in the type of the indexing variable. However, for the sequence operator, we require this type to be a finite sequence. As expected, the process $\mathbin{;}\ x : T \bullet P(x)$ is the sequential composition of processes $P(v)$, with v taken from T in the order that they appear.

Circus introduces a new operator that can be used to define processes. The indexed process $i : T \odot P$ behaves exactly like P, but for each channel c of P, we have a freshly named channel c_i. These channels are implicitly declared by the indexed operator, and communicate pairs of values: the first element, the index, is a value i of type T, and the second element is the value of the original type of the channel. An indexed process P can be instantiated using the instantiation operator $P\lfloor e \rfloor$; it behaves just like P, however, the value of the expression e is used as the first element of the pairs communicated through all the channels.

For instance, we may define a process similar to the previously defined $RegisterTwice$, in order to have the same process that represents two $Register$s running independently, but with an identification of which process is reset. In order to interact with the indexed process $IndexRegister \mathrel{\widehat=} i : \{1, 2\} \odot Register$, we must use the channels $store\_i$, add_i, $result\_i$, out_i and $reset\_i$. We may instantiate the process $IndexRegister$: the process $IndexRegister\lfloor 1 \rfloor$, for instance, outputs pairs through channel out_i whose first elements are 1 and the second elements are the values stored in the register. It may be restarted by sending the value 1 through the channel $reset\_i$. Similarly, we have the process $IndexRegister\lfloor 2 \rfloor$. Finally, we have the process presented below that represents a pair of registers: the first element of the pairs identifies the register.

$$RegisterTwiceId \mathrel{\widehat=} IndexRegister\lfloor 1 \rfloor \interleave IndexRegister\lfloor 2 \rfloor$$

The renaming operator $P[oldc := newc]$ replaces all the communications that are done through channels $oldc$ by communications through channels $newc$, which are implicitly declared, if needed. Usually, indexing and renaming are used in conjunction, as in the redefinition of the process $RegisterTwice$ presented below.

$$RegisterTwice \mathrel{\widehat=} RegisterTwiceId \left[\begin{array}{lcl} store\_i, add\_i, & & storeid, addid, \\ result\_i, out\_i, & := & resultid, outid, \\ reset\_i & & resetid \end{array} \right]$$

We may also combine instantiations of an indexed process using the iterated operators. For example, we may redefine the process $RegisterTwiceId$ as $\interleave\ i : \{1, 2\} \bullet Register\lfloor i \rfloor$. The same characteristics and restrictions still apply to the iterated operators.

Finally, generic processes may be instantiated: the expression $P[T]$ instantiates a generic process named P using the type T.

2.1.6 Actions

When a process is explicitly defined, besides the definitions of the state and the main action, we have in its body Z paragraphs, definitions of (parametrised) actions, and variable sets definitions; they are used to specify the main action of the process.

$$\mathsf{PPar} \quad ::= \quad \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{ParAction} \mid \textbf{nameset}\ \mathsf{N} == \mathsf{NSExp}$$

Like channel sets, the empty set $\{\}$, variable name enumerations enclosed in $\{$ and $\}$, and expressions formed by some of the Z set operators are the elements of the syntactic category NSExp.

As with processes, an action may be parametrised, in which case we have the declaration of the parameters followed by a $\bullet$, and then, the body of the action.

$$\mathsf{ParAction} ::= \quad \mathsf{Action} \mid \mathsf{Decl} \bullet \mathsf{ParAction}$$

An action can be a schema expression, a guarded command, an invocation to a previous defined action, or a combination of these constructs using CSP operators. Furthermore, state components and local variables may be renamed; however, no channel name can be changed.

$$\mathsf{Action} \quad ::= \quad \mathsf{SchemaExp} \mid \mathsf{Command} \mid \mathsf{N} \mid \mathsf{CSPAction} \mid \mathsf{Action}\,[\mathsf{N}^{+} := \mathsf{Exp}^{+}]$$

Three primitive actions are available in *Circus*: *Skip*, *Stop*, and *Chaos*. The action *Skip* does not communicate any value or changes the state: it terminates immediately. The action *Stop* deadlocks, and the action *Chaos* diverges.

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

The syntactic category Pred is that of Z predicates defined in [91], which is supported by the *Circus* parser that is currently available.

The prefix operator is standard. However, a guard construction may be associated with it. For instance, given a Z predicate p, if the condition p is $true$, the action $p\ \&\ c?x \rightarrow A$ inputs a value through channel c and assigns it to the variable x, and then behaves like A, which has the variable x in scope. If, however, the condition p is $false$, the same action deadlocks. Such enabling conditions like p may be associated with any action. Predicates may also be associated with an input prefix. For instance, a communication $c?x : p$ will only happen when a value of the type of the channel c that satisfies the predicate p is communicated.

The action *Sum* in the process *SumClient* (Figure 2.1) exemplifies the output prefix operator. While the number n is different from 0, this action requests the *Register* to add a value to its current value by outputting n through channel *add*. Finally, when n reaches 0, it requests the *result* from the *Register*, reads it from channel *out*, and writes it to channel *write*.

All the free variables of an action must be in scope in the containing process. All actions are in the scope of the state components. Input communications introduce new variables into scope, which may not be used as targets of assignments.

The CSP operators of sequence, external and internal choice, parallel, interleaving, and hiding may also be used to compose actions. However, differently from processes, at the level actions, recursive definitions are also available (μ).

Our *Register*, as previously described, has a recursive behaviour. Its cycle, the action *RegCycle*, is an external choice: values may be stored or accumulated, using channels *store* and *add*; the result may be requested using channel *result*, and output through *out*; finally, the register may be reset through channel *reset*.

At the level of actions, the parallel and the interleaving operators are slightly different from that of CSP in [80] and [52]. In order to avoid conflicts in the access to the variables in scope, parallel composition and interleaving of actions must also declare two disjoint sets (that may partition) of variables in scope: state components, and input and local variables. In $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, both A_1 and A_2 have access to the initial values of all variables in ns_1 and ns_2, but A_1 may modify only the values of the variables in ns_1, and A_2, the values of the variables in ns_2. Besides, the actions A_1 and A_2 synchronise on the channels in the set cs.

Parametrised actions can be instantiated: for instance, we can have the action $A(x)$, if A is a previously defined single-parametrised action; we can also have an instantiation of the form $(x : T \bullet A)(x)$.

As for processes, the iterated operators for sequence, external and internal choice, parallel, and interleaving can also be used in order to generalise the corresponding operators.

Actions may also be defined using Dijkstra's guarded commands [37].

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

An action can be a (multiple) assignment, or a guarded alternation. For instance, we store a value in the *Register* using the assignment $value := newValue$. Variable blocks can also be used in an action specification. In the interest of supporting a calculational approach to development, an action can also be written as a specification statement in the style of Morgan's refinement calculus [63]. We adopt the syntactic sugaring $\{pre\}$ for specification statements $: [pre, true]$ (assumptions). In the same way, the coercion $[post]$ is a syntactic sugaring for $: [true, post]$. The invocation of substitutions by value, result, or by value-result, as those presented in [22], are also available in *Circus*.

2.2 The *Unifying Theories of Programming*

The semantic model of *Circus* is based on Hoare & He's *Unifying Theories of Program-*

ming [54]. The UTP is a framework in which the theory of relations is used as a unifying basis for programming science across many different computational paradigms: procedural and declarative, sequential and parallel, closely-coupled and distributed, and hardware and software. All programs, designs, and specifications are interpreted as relations between an initial observation and a single subsequent observation, which may be either an intermediate or a final observation, of the behaviour of program execution.

Common ideas, such as sequential composition, conditional, nondeterminism, and parallel composition are shared by different theories of different programming paradigms. For instance, sequential composition is relational composition, conditional is boolean connective, nondeterminism is disjunction, and parallel composition is a restricted form of conjunction. Miracle is interpreted as an empty relation, abortion is interpreted as the universal relation, and correctness and refinement is interpreted as inclusion of relations: reverse implication. All the laws of the relational calculus may be used for reasoning about correctness in all theories and in all languages.

Three elements of a theory are used to differentiate different programming languages and design calculi: the alphabet, a set of names that characterise a range of external observations of a program behaviour; the signature, which provides syntax for denoting the objects of the theory; and the healthiness conditions, which select the objects of a sub-theory from those of a more expressive theory in which it is embedded.

The alphabet of a theory collects the names within the theory that identify observation variables that are important to describe all relevant aspects of a program behaviour. The initial observations of each of these variables are undecorated and compose the input alphabet ($in\alpha$) of a relation. Subsequent observations are decorated with a dash and compose the output alphabet ($out\alpha$) of a relation. This allows a relation to be expressed as in Z by its characteristic predicate. Table 2.1 summarises the observational variables of the UTP that are used in the semantics of *Circus*.

In *Circus*, some combinations of these variables have interesting semantic meaning. For instance, $okay' \wedge wait'$ represents a non-divergent state of a process that is waiting for some interaction with the environment; if, however, we have $okay' \wedge \neg\ wait'$, the non-divergent process has terminated; finally, $\neg\ okay'$ represents a divergent process.

Besides these variables, there are also UTP theories that include variables that may be used to represent program control, real time clock, or resource availability. For each theory, we may select a subset of relevant variables.

The signature of a theory is a set of operators and atomic components of this theory: it is the syntax of the language. The smaller the signature, the simpler the proof techniques to be applied for reasoning. Signatures may vary according to the theory's purpose. Specification languages are least restrictive and often include quantifiers and all relational calculus operators. Design languages successively remove non-implementable operators. The negation is the first one to be removed. Thus, all operators are monotonic, and recursion can safely be introduced as a fixed-point operator. Finally, programming languages present only implementable operators in their signature. They are commonly defined in terms of their observable effects using the more general specification language.

Healthiness conditions are used to test a specification or design for feasibility, and reject it if it makes implementation impossible in the target language. They are expressed

okay	This boolean variable indicates if the system has been properly started in a stable state, in which case its value is *true*, or not; $okay'$ means subsequent stabilisation in an observable state.
tr	This variable, whose type is a sequence of events, records all the events in which a program has engaged.
wait	This boolean variable distinguishes the intermediate observations of waiting states from final observations on termination. In a stable intermediate state, $wait'$ has *true* as its value; a *false* value for $wait'$ indicates that the program has reached a final state.
ref	This variable describes the responsiveness properties of the process; its type is a set of events. All the events that may be refused by a process before the program has started are elements of *ref*, and possibly refused events at a later moment are referred by ref'.
v	All program variables (state components, input and local variables, and parameters) are collectively denoted by v.

Table 2.1: *Circus* Alphabet

in terms of an idempotent function ϕ that makes a program healthy. Every healthy program P must be a fixed-point $P = \phi(P)$. Some healthiness conditions are used to identify the set of relations that are designs (**H1** and **H2**), reactive processes (**R1**-**R3**), and CSP processes (**CSP1**-**CSP2**). In Chapter 3.1, we discuss the relevant ones in more detail.

In Figure 2.2, we present how some of the theories presented in [54] are related. Relations are predicates with an input and an output (dashed) alphabet. Designs are relations that are **H1** and **H2** healthy. Reactive processes are **R1**, **R2** and **R3** healthy relations (this composition is represented by the healthiness condition **R**). Finally, there are two ways of characterising the CSP processes: they are characterised as reactive processes that are **CSP1** and **CSP2** healthy, or as relations that result from applying **R** to designs.

2.3 Final Considerations

Circus has been suggested as a link between two different schools of formal methods for software engineering: the state-based school and the process algebraic. The former is strongly represented by VDM [57], Z [107], and B [3], and the latter is strongly represented by CCS [62] and CSP [52, 80]. Besides providing a link between these two schools, *Circus* also includes a refinement theory and a refinement strategy in a calculational style as in [63]. A refinement strategy for *Circus*, based on laws of simulation, and action and process refinement (Appendix C), has been proposed in [27] and is also extended in this thesis.

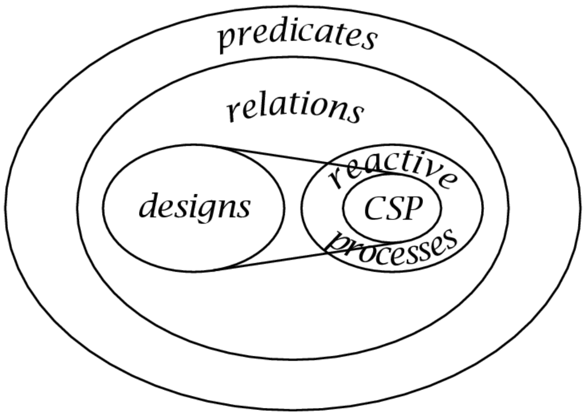


Figure 2.2: Theories in the UTP

Basically, *Circus* programs are characterised by processes, which group paragraphs that describe data and control behaviour. Mainly, we use the Z notation [91] to define data, and actions, which are defined using Z, CSP, and guarded commands constructs, to characterise behaviour.

Research involving *Circus* has been taken into a wide range of areas. In [45], a model-checker for *Circus* and its theoretical foundations [106] are presented. In [86], Sherif and He propose a time model for *Circus* and present a simple case study. A denotational semantics for mobile processes in Hoare & He's *Unifying Theories of Programming* can be found in [94]; this is the first step towards a mobile *Circus*. Object-orientation is also being considered by Sampaio, Woodcock and Cavalcanti [28, 15], and a mapping from UML to *Circus* specifications is also under research [24]. Xavier is investigating a type-checker for *Circus*. An automatic translation from *Circus* to JCSP that implements the strategy presented in Chapter 6 can be found in [73]. Furthermore, synchrony, testing, *Circus* compliance [5], Control Law Diagrams [23], and Ravenscar [6] are also in the *Circus* agenda of research.

Chapter 3

Circus Denotational Semantics

This chapter presents the *Circus* denotational semantics and the first step towards a theorem prover for *Circus*: the mechanisation of the *Circus* semantics in a theorem prover, ProofPower-Z. First, Section 3.1 presents the *Circus* denotational semantics, which is based on the UTP. Finally, in Section 3.2 we discuss the mechanisation of part of the UTP and *Circus*. Most of the material presented in Section 3.2 was published in [77].

3.1 *Circus* Denotational Semantics

A denotational semantics for *Circus* was first published in [105], where Cavalcanti and Woodcock base their work on the UTP; their model for a *Circus* program is a Z specification. By using Z, their semantics allowed the use of tools like Z/EVES [83] to analyse and validate their definitions, and to reason about systems specified in *Circus*. Unfortunately, that semantics is not appropriate to prove our refinement laws. The reason is that in [105] the authors provided a shallow embedding of *Circus* in Z; however, in order to prove properties about *Circus* itself, like our refinement laws, a deep embedding of *Circus* in Z is needed. The denotational semantics of *Circus* that we present in the sequel is based on the work presented in [105] and [54], and constitutes the definitive reference to the *Circus* denotational semantics. The mechanisation of the semantics is a conservative extension of the existing theories of ProofPower-Z, which, by themselves, are defined as conservative extensions. Because these are all conservative extensions, this guarantees soundness.

In [31], Cavalcanti and Woodcock present an introduction to CSP in the UTP. Their definitions correspond to the ones presented in [54], but with a different style of specification: every CSP process is defined as a reactive design of the form $\mathbf{R}(pre \vdash post)$. A design $pre \vdash post$ is defined as $okay \land pre \Rightarrow okay' \land post$: if the program starts in a state satisfying its *pre*condition, the design will terminate, and, on termination, it will establish its *post*condition. Using this style, we use a design to define the behaviour of a process when its predecessor has terminated and not diverged; the process behaviour in the other situations is defined by the healthiness condition $\mathbf{R}$, which, as discussed in Chapter 2, is a composition of the three healthiness condition presented in Table 3.1.

	Formal Representation	Description
R1	$\mathbf{R1}(P) \mathrel{\widehat=} P \wedge tr \leq tr'$	The execution of a reactive process never undoes any event that has already been performed.
R2	$\mathbf{R2}(P(tr, tr')) \mathrel{\widehat=} P(\langle\rangle, tr' - tr)$	The behaviour of a reactive process is oblivious to what has gone before.
R3	$\mathbf{R3}(P) \mathrel{\widehat=} \mathit{II}_{rea} \lhd wait \rhd P$	Intermediate stable states do not progress.

Table 3.1: Healthiness Conditions — Reactive Processes

The first healthiness condition, **R1**, states that the history of interactions of a process cannot be changed, therefore, the value of tr can only get longer. The condition $tr \leq tr'$ holds if, and only if, the sequence tr is a prefix of or equal to the sequence tr'. The second healthiness condition, **R2**, establishes that a reactive process should not rely on the interactions that happened before its activation. The expression $s - t$ stands for the result of removing an initial copy of t from s; this partial operator is only well-defined if t is a prefix of s. The sequence $tr' - tr$ represents the traces of events in which the process itself has engaged from the moment it starts to the moment of observation. The final healthiness condition, **R3**, defines the behaviour of a process that is still waiting for another process to finish: it should not start. If the condition b is true, the predicate $P \lhd b \rhd Q$ is equivalent to P; otherwise, it is equivalent to Q. Formally, it is defined as $(b \wedge P) \vee (\neg\, b \wedge Q)$.

In [54] it is not quite clear whether CSP processes may have state or not; however, it is clear that, if there are state variables, they are not changed. In our work, we consider the state variables as part of the following definition for the reactive skip.

Definition B.1

$$\begin{aligned} \mathit{II}_{rea} \mathrel{\widehat=}\ & (\neg\, okay \wedge tr \leq tr') \\ & \vee (okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v) \end{aligned}$$

If the previous process diverged, the reactive skip only guarantees that the history of communication is not forgotten; otherwise, it terminates and keeps the values of the variables unchanged. For conciseness, throughout this chapter, given a process with state components and local variables $x_1, \ldots, x_n$, the predicate $v' = v$ denotes the conjunction $x_1' = x_1 \wedge \ldots \wedge x_n' = x_n$.

In what follows, we take the approach of [31]: a vast majority of the *Circus* actions are defined as reactive designs of the form $\mathbf{R}(pre \vdash post)$. Those which are not defined in this way, reuse the results of [54] and were proved to be indeed reactive. As a direct consequence of this, we have that the following theorem holds; its proof is by induction on the structure of the *Circus* actions.

Theorem 3.1 *Every Circus action is **R** (**R1**, **R2**, and **R3**) healthy.*

We start this section by giving semantics to CSP actions in Section 3.1.1. In Section 3.1.2, we discuss the semantics of action invocation, parametrised actions and renaming. The semantics of *Circus* commands and schema expressions are presented in Sections 3.1.3 and 3.1.4, respectively. In Section 3.1.5, we present and discuss the semantics of *Circus* processes. Finally, we present further healthiness conditions which are satisfied by every *Circus* program.

3.1.1 CSP Actions

The first action we present is the deadlock action *Stop*: it is incapable of engaging in any events and is always waiting.

Definition B.2 $Stop \mathrel{\widehat=} \mathbf{R}(true \vdash tr' = tr \land wait')$

Stop has a *true* precondition because it never diverges. Furthermore, it never engages in any event and is indefinitely waiting; therefore, its trace is left unchanged and $wait'$ must be true. Since it represents deadlock, *Stop* must refuse all events. We express this by leaving the final value of the refusal set, ref', unconstrained; any refusal set is a valid observation. Since state changes do not decide the choice, as we explain later in this section, *Stop* must leave the values of the state components unconstrained in order to be the unit for the external choice (see Section 4.5 for details).

Skip is the action that terminates immediately and makes no changes to the trace or to the state components.

Definition B.3 $Skip \mathrel{\widehat=} \mathbf{R}(true \vdash tr' = tr \land \neg\, wait' \land v' = v)$

The value of ref' is left unspecified because it is irrelevant after termination.

The worst *Circus* action is *Chaos*; it has an almost unpredictable behaviour.

Definition B.4 $Chaos \mathrel{\widehat=} \mathbf{R}(false \vdash true)$

Since it is defined as a reactive design, *Chaos* cannot undo the events of a process history. For this reason, it is not the right zero for sequential composition. The sequential composition P; *Chaos* only diverges after the successful termination of P. For instance, the sequential composition $(c \rightarrow Skip)$; *Chaos* only diverges after the synchronisation on c; however, the definition above guarantees that c is in the final trace of the sequential composition, whereas *Chaos* alone only guarantees that the initial trace is a prefix of the final trace ($tr \leq tr'$).

Circus sequential composition is trivially defined as relational sequence, which is explained in detail in Section 3.2.3. The guarded action $g \;\&\; A$ behaves like *Stop* if g is false, and like A otherwise. For conciseness, in the definition that follows and throughout this chapter, we abbreviate $A[b/okay'][c/wait]$ as A^b_c. Basically, A^f_f gives us the conditions in which action A diverges when it is not waiting for its predecessor to finish, and A^t_f gives

	Formal Representation	Description
CSP1	$\mathbf{CSP1}(P) \mathrel{\widehat=} P \lor (\neg\, okay \land tr \le tr')$	Extension of the trace is the only guarantee on divergence
CSP2	$\mathbf{CSP2}(P) \mathrel{\widehat=} P; J$	A process may not require non-termination
CSP3	$\mathbf{CSP3}(P) \mathrel{\widehat=} SKIP; P$	A process does not depend on *ref*

Table 3.2: Healthiness Conditions — CSP Processes

us the conditions that are satisfied when A terminates without diverging.

Definition B.6 $g \mathbin{\&} A \mathrel{\widehat=} \mathbf{R}((g \Rightarrow \neg\, A^f_f) \vdash ((g \land A^t_f) \lor (\neg\, g \land tr' = tr \land wait')))$

If the guard g is *false*, this definition can be reduced to *Stop*. However, if the guard g is *true*, we are left with the reactive design $\mathbf{R}(\neg\, A^f_f \vdash A^t_f)$; the following theorem (from [54]) shows us that this reactive design is exactly A itself.

Theorem 3.2 *For every CSP process A,* $A = \boldsymbol{R}(\neg\, A^f_f \vdash A^t_f)$.

This theorem is proved in [31] and applies to CSP processes. These processes are defined in the UTP as reactive designs that satisfy two other healthiness conditions presented in Table 3.2: the only guarantee on divergence of a **CSP1** process is the extension of the trace, and **CSP2** processes may not require non-termination. In the definition of **CSP2** we take the approach of [31] instead of that in [54]. We make use of an idempotent function **CSP2**, which is defined in terms of a predicate J defined as follows:

$$J \mathrel{\widehat=} (okay \Rightarrow okay') \land tr' = tr \land wait' = wait \land ref' = ref \land v' = v$$

Besides **CSP1** and **CSP2**, processes that can be defined using the notation of CSP satisfy other healthiness conditions. One of them, **CSP3**, requires that the behaviour of a process does not depend on the initial value of *ref*.

The following theorem guarantees that *Circus* actions are indeed **CSP1**, **CSP2** and **CSP3** healthy, and therefore, Theorem 3.2 is applicable to them.

Theorem 3.3 *Every Circus action is* ***CSP1****,* ***CSP2****, and* ***CSP3*** *healthy.*

Part of the proof of this theorem is a direct result from the fact that reactive designs are indeed **CSP1** and **CSP2** [31]. The rest of the proof is done by induction on the syntax of the language; for the sake of conciseness, it is omitted here. This proof and the proof of all the new theorems presented in this chapter can be found in [71].

When its predecessor has terminated without diverging, an external choice $A_1 \mathbin{\Box} A_2$ does not diverge if neither A_1 nor A_2 do. We capture this behaviour in the precondition of the following definition of external choice. The postcondition of this reactive design establishes that if the trace has not changed and the choice has not terminated, the behaviour of an external choice is given by the conjunction of the effects of both actions;

otherwise, the choice has been made and the behaviour is either that of A_1 or A_2.

Definition B.7

$$A_1 \,\Box\, A_2 \mathrel{\widehat{=}} \mathbf{R}((\neg\, {A_1}_f^f \wedge \neg\, {A_2}_f^f) \vdash (({A_1}_f^t \wedge {A_2}_f^t) \lhd tr' = tr \wedge wait' \rhd ({A_1}_f^t \vee {A_2}_f^t)))$$

It is a direct and important consequence of this definition that a state change does not resolve a choice; this would be expressed by including $v' = v$ in the condition of the postcondition. By way of illustration, let us consider the following choice.

$$(x := 0;\; c_1 \rightarrow Skip) \,\Box\, (x := 1;\; c_2 \rightarrow Skip)$$

This choice does not happen instantly; it only happens when either c_1 or c_2 happens. The final value of x depends on which communication actually happens. We have chosen state changes not to resolve an external choice because states are encapsulated within a *Circus* process, and so their changes should not be noticed by the external environment.

The internal choice is the first constructor which is not defined as a reactive design: it is simply the disjunction of both actions.

Definition B.8 $A_1 \sqcap A_2 \mathrel{\widehat{=}} A_1 \vee A_2$

This is a simple definition, and the use of reactive designs to define an internal choice gives rise to a slightly more complicated definition; for this reason, we keep the disjunction. In fact, if we consider A_1 and A_2 to be $\mathbf{R}(pre_1 \vdash post_1)$ and $\mathbf{R}(pre_2 \vdash post_2)$, respectively, the following theorem holds.

Theorem 3.4 $A_1 \sqcap A_2 \mathrel{\widehat{=}} \boldsymbol{R}(pre_1 \wedge pre_2 \vdash post_1 \vee post_2)$

An internal choice diverges if either of the preconditions is not valid and establishes either $post_1$ or $post_2$ on termination.

Because we express it as a reactive design, our semantics for prefix is simpler than the one presented in [54]. It uses the function $do_{\mathcal{C}}$ presented below, which gives the behaviour of the prefix regarding the observational variables tr and ref. For us, an event is a pair (c, e), where the first element is the name of the channel and the second element is the value which was communicated. For synchronisation events, we have the special value $Sync$.

Definition B.9 $do_{\mathcal{C}}\,(c, e) \mathrel{\widehat{=}} tr' = tr \wedge (c, e) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (c, e) \rangle$

While waiting, an action that is willing to synchronise on an event (c, e) has not changed its trace and cannot refuse this event. After the communication ($\neg\, wait'$), the event is included in the trace of the action.

A synchronisation $c \rightarrow Skip$ does not diverge; neither does it change the state.

Definition B.10 $c \rightarrow Skip \mathrel{\widehat=} \mathbf{R}(true \vdash do_{\mathcal{C}}(c, Sync) \wedge v' = v)$

In *Circus*, output communications are a simply syntactic sugaring for synchronisations on output values. The only difference between a synchronisation event and a synchronisation in some value is that the communicated value is taken into account.

Definition B.11 $c.e \rightarrow Skip \mathrel{\widehat=} \mathbf{R}(true \vdash do_{\mathcal{C}}(c, e) \wedge v' = v)$

In fact, for any communication that does not involve input, we have the following definition.

Definition B.13 For any non-input communication, $c \rightarrow A \mathrel{\widehat=} (c \rightarrow Skip);\ A$.

This definition presents a way of expressing an action A prefixed by a communication c as a sequential composition of the communication c and A.

Input prefix has a slightly more complex definition. This is because we must consider every possible value that can be communicated through the given channel. Besides, once the communication happens, the value of the input variable changes accordingly. The function $do_{\mathcal{C}}$ presented above does not consider these facts; we present another function, $do_{\mathcal{I}}$, which although similar to $do_{\mathcal{C}}$, takes these aspects into account. In the following definition, we consider the availability of an environment δ, that gives us the types of every channel in the system. Before the communication, an input prefix $c?x : P$ cannot refuse any communication on a set of acceptable events; these are the events on c that communicate values of the type of c which satisfy the predicate P. After the communication the trace is incremented by one of these possible events. Besides, the final value of x is that which was communicated. The function *snd* returns the second element of a pair, and the function *last* returns the last element of a non-empty list.

Definition B.14

$$
\begin{aligned}
do_{\mathcal{I}}(c, x, P) \mathrel{\widehat=}\ & tr' = tr \wedge \{e : \delta(c) \mid P \bullet (c, e)\} \cap ref' = \varnothing \\
& \lhd wait' \rhd \\
& tr' - tr \in \{e : \delta(c) \mid P \bullet \langle (c, e) \rangle\} \wedge x' = snd(last(tr'))
\end{aligned}
$$

In the same way we did for non-input prefix, we define the input prefix in terms of the function $do_{\mathcal{I}}$ above. However, an input prefix $c?x : P \rightarrow A(x)$ implicitly declares a new variable x and, after the communication, uses the communicated value in A. In the following definition we declare the new variable x using a *Circus* variable block whose semantics will be presented later in this section.

Definition B.15 $c?x : P \rightarrow A(x) \mathrel{\widehat=} \textbf{var}\ x \bullet \mathbf{R}(\mathit{true} \vdash do_{\mathcal{I}}(c, x, P) \wedge v' = v);\ A(x)$

The predicate *true* may be omitted in an input prefix.

An interesting and helpful theorem is presented below. It allows us to express an input prefix in terms of an external choice, provided the set of values that can be communicated is finite. This theorem makes the proofs of some refinement laws much simpler for finite channels.

Theorem 3.5 $c?x : P \rightarrow A(x) \mathrel{\widehat=} \ \Box\, x : \{e : \delta(c) \mid P\} \bullet c.x \rightarrow A(x)$, *provided* $\{e : \delta(c) \mid P\}$ *is finite.*

In this thesis, we do not consider all the possible combinations of inputs and outputs that can be used in a channel of infinite type. Their semantics is lengthy, but not illuminating. For conciseness, we omit the definition of combinations of inputs and outputs that can be used in a channel of finite type; all the definitions can be found in Appendix B. For channels with a finite type, we consider the Theorem 3.5 to transform these possible combinations into an external choice of simple synchronisations.

The parallel composition $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$ models interaction and synchronisation between the two concurrent actions A_1 and A_2. Another consideration for our semantics is that we assume that the references to names and channels sets have already been expanded using their corresponding definitions. As explained in Chapter 2, in *Circus*, we follow the alphabetised parallel composition adopted by [80]: only events that are in the specified synchronisation channel set cs are required to happen simultaneously in both A_1 and A_2; the remaining events may happen independently. In what follows, we present the semantics of parallel operator as a reactive design in two parts: first we discuss its precondition, and then, we discuss its postcondition.

Divergence can only happen if it is possible for either of the actions to reach divergence. This can be expressed by trying to find a trace that leads one of the actions to divergence and on which both actions agree regarding cs. For instance, the following expression tells us if it is possible for A_1 to diverge.

$$\exists\, 1.tr', 2.tr' \bullet ({A_1}^f_f;\ 1.tr' = tr) \wedge ({A_2}_f;\ 2.tr' = tr) \wedge 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \qquad [\text{P1}]$$

Basically, if there exist two traces $1.tr'$ and $2.tr'$, defined as a trace of A_1 after divergence and as a trace of A_2, and if these two traces are equal modulo cs, then it is possible for A_1 to reach divergence. First, we define the trace $1.tr'$ on which A_1 diverges as ${A_1}^f_f;\ 1.tr' = tr$. The first predicate of the sequence give us the conditions on which A_1 diverges; we record the final trace in $1.tr'$ in the second predicate of the sequence, which ignores the final values of the other variables. Similarly, we define $2.tr'$ for A_2 as ${A_2}_f;\ 2.tr' = tr$. Since we are not interested in divergence, we do not replace $okay'$ by any particular value. Finally, we compare both traces using the sequence filtering function $\upharpoonright$; given a sequence sq and a set st, $sq \upharpoonright st$ gives us the largest subsequence of sq containing only those objects that are elements of st.

In a very similar way as we presented above for A_1, we can also express the possibility of divergence in A_2. The parallel composition diverges if either of these two conditions is true; hence, the precondition of the reactive design for the parallel composition is the conjunction of the negation of both conditions.

$$
\begin{array}{l}
\neg\ \exists\, 1.tr', 2.tr' \bullet ({A_1}^f_f;\ 1.tr' = tr) \land ({A_2}_f;\ 2.tr' = tr) \land 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\
\land\ \neg\ \exists\, 1.tr', 2.tr' \bullet ({A_1}_f;\ 1.tr' = tr) \land ({A_2}^f_f;\ 2.tr' = tr) \land 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs
\end{array}
$$

For the postcondition, we use the parallel by merge technique used by Hoare and He in the UTP. Basically, we run both actions independently and merge their results afterwards.

$$
(({A_1}^t_f;\ U1(out\alpha\, A_1)) \land ({A_2}^t_f;\ U2(out\alpha\, A_2)))_{+\{v,tr\}};\ M_{\|_{cs}}
$$

In order to express their independent executions, we use relabelling function Ul: the result of applying Ul to an output alphabet $\{v_1', \ldots, v_n'\}$ is the predicate presented below.

$$
l.v_1' = v_1 \land \ldots \land l.v_n' = v_n
$$

Before the merge, however, we extend the alphabet of the predicate presented above that expresses the independent execution of both actions. For a predicate P and name n, the alphabet extension $P_{+\{n\}}$ is equivalent to the predicate $P \land n' = n$. By extending the alphabet with v' and tr' in the above definition, we record the initial values of the trace tr and of the state components and local variables v in tr' and v', respectively; they can be used by the merge function $M_{\|_{cs}}$, as we explain in the sequel.

The merge function $M_{\|_{cs}}$ is not only responsible for merging the traces of both action, but also for merging the state components, local variables and the remaining UTP observational variables.

$$
\begin{array}{rl}
M_{\|_{cs}} \mathrel{\widehat=} & tr' - tr \in (1.tr - tr \parallel_{cs} 2.tr - tr) \land 1.tr \upharpoonright cs = 2.tr \upharpoonright cs \\
\land & \left(\begin{array}{l}
\left(\begin{array}{l}
(1.wait \lor 2.wait) \land \\
ref' \subseteq ((1.ref \cup 2.ref) \cap cs) \cup ((1.ref \cap 2.ref) \setminus cs)
\end{array}\right) \\
\lhd wait' \rhd \\
(\neg\ 1.wait \land \neg\ 2.wait \land MSt)
\end{array}\right)
\end{array}
$$

The trace $(tr' - tr)$ is extended according to the merge of the new events that happened in both actions. The function $\parallel_{cs}$ takes each of the individual traces and gives a set containing all the possible combinations of these two traces according to cs. Its definition is omitted here for conciseness, but can be found in Appendix B, and is originally presented in [80]. The expression before the merge gives us all the possible behaviours of running A_1 and A_2 independently; however, only those combinations that are feasible regarding the synchronisation on cs should be considered. We eliminate the combinations that are not feasible by including the restriction that the traces must be equal modulo cs. Finally, the parallel composition has not terminated if any of the actions have not terminated. In this case, the parallel composition refuses all events in cs that are being refused by any of the actions and all the events not in cs which are being refused by both actions. In order to terminate, both actions in the parallel composition must terminate; in this case, we merge the state as follows.

$$\begin{aligned} MSt \mathrel{\widehat=} \; & \forall v \bullet (v \in ns_1 \Rightarrow v' = 1.v) \land (v \in ns_2 \Rightarrow v' = 2.v) \\ & \land (v \notin ns_1 \cup ns_2 \Rightarrow v' = v) \end{aligned}$$

For every local variable and state component v, if it is declared in ns_1, its final value is that of A_1; if, however, it is declared in ns_2, its final value is that of A_2. Finally, if it is declared in neither ns_1 nor ns_2, its value is left unchanged.

We present below the whole of the semantics of parallel composition.

Definition B.18

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 \mathrel{\widehat=}$$
$$\mathbf{R}\left(\begin{array}{l} \neg\ \exists 1.tr', 2.tr' \bullet ({A_1}^f_f;\ 1.tr' = tr) \land ({A_2}_f;\ 2.tr' = tr) \\ \qquad\qquad\qquad\qquad \land\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\ \land\ \neg\ \exists 1.tr', 2.tr' \bullet ({A_1}_f;\ 1.tr' = tr) \land ({A_2}^f_f;\ 2.tr' = tr) \\ \qquad\qquad\qquad\qquad \land\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\ \vdash \\ (({A_1}^t_f;\ U1(out\alpha\, A_1)) \land ({A_2}^t_f;\ U2(out\alpha\, A_2)))_{+\{v,tr\}};\ M_{\parallel_{cs}} \end{array}\right)$$

The semantics of interleaving does not have to consider any synchronisation channel. An interesting aspect regarding the differences between the definitions of parallel composition and interleaving is the much simpler precondition for interleaving. Since both actions may execute independently, the interleaving of two actions diverges if either of the actions do so. Therefore, its precondition is the same as that for external choice $\neg\ {A_1}^f_f \land \neg\ {A_2}^f_f$. Its postcondition is very similar to that of parallel operator, but uses a different merge function $M_{\interleave_{cs}}$. As a matter of fact, interleaving is equivalent to parallel composition on an empty synchronisation channel set; this is stated by the refinement Law C.98.

As was the case with internal choice, the hiding operator is not defined as a reactive design. The calculations to express hiding as a reactive design pointed out that the final definition would be quite complicated and extensive; hence, we preferred to base our definition on that presented in [54] for the CSP hiding.

Definition B.20

$$\begin{aligned} & A \setminus cs \mathrel{\widehat=} \\ & \quad \mathbf{R}(\exists s \bullet A[s, cs \cup ref'/tr', ref'] \land (tr' - tr) = (s - tr) \upharpoonright (EVENT - cs));\ Skip \end{aligned}$$

If A reaches a stable state in which it cannot perform any further events in cs, than the action $A \setminus cs$ has also reached such state. The new events $(tr' - tr)$ performed by $A \setminus cs$ are those new events performed by A (in Definition B.20, we rename the final trace of A to s; so $s - tr$ gives us the new events of A), but filtered by the set of all events but those in cs. We also include the events in cs in the final refusal set of A by replacing ref' by $cs \cup ref'$. *Skip* guarantees that possible divergences introduced by hiding events in a recursive action (Law C.131) are actually captured. The proof of Law C.131, which can be found in [71], illustrates this situation. The calculation of the left-hand side of the sequence leaves us with the predicate $\mathbf{R}(tr' = tr)$; *Skip*; it is the *Skip* on the right-hand side of the sequence that allows us to reduce this sequential composition to *Chaos*.

A recursive action can be expressed in two ways: it can be explicitly defined using the weakest fixed-point ($\mu X \bullet F(X)$), or it can be implicitly defined by invoking the action itself. For instance, we present below two ways of expressing a recursive action A that indefinitely performs the event c.

$$A \mathrel{\widehat=} \mu X \bullet c \to X \qquad \text{[Explicit notation]}$$
$$A \mathrel{\widehat=} c \to A \qquad \text{[Implicit Notation]}$$

The transformation from one notation to another is purely syntactic; the implicit notation is simply syntactic sugar for the explicit one. In this chapter, we consider only the first notation. The semantics of the action $\mu X \bullet A(X)$ is standard: for a monotonic function F from *Circus* actions to *Circus* actions, the weakest fixed-point is defined as the greatest lower bound (the *weakest*) of all the fixed-points of F. In the definition below, $\sqsubseteq_{\mathcal{A}}$ stands for action refinement; its definition, which is expressed as an inverse implication, can be found in Section 4.1.

Definition B.21 $\mu X \bullet F(X) \mathrel{\widehat=} \bigsqcap \{X \mid F(X) \sqsubseteq_{\mathcal{A}} X\}$

The iterated operators are used to generalise the binary operators of sequence, external and internal choice, parallel composition, and interleaving; only finite types can be used for the indexing variables. Basically, the semantics of all the iterated operators is given by the expansion of the operator. For sequence, we have that the type of the indexing variables are finite sequences; the expansion respects this sequence.

Definition B.22 $\mathbin{⨾}\, x : \langle v_1, \ldots, v_n \rangle \bullet A(x) \mathrel{\widehat=} A(v_1);\ \ldots;\ A(v_n)$

The definitions of the other iterated operators are very similar. However, in the expansion of iterated parallel composition and interleaving, the state partitions, which are also parametrised by the indexing variable, must be considered. For example, given a set of channels cs, a function f from numbers ranging in the interval $0 \mathinner{..} 2$ to state components, and a parametrised action A, the definition of the iterated parallel composition $\llbracket\, cs \,\rrbracket\, x : 0 \mathinner{..} 2 \bullet \llbracket f(x) \rrbracket\, A(x)$ is $A(0) \llbracket f(0) \mid cs \mid f(1) \cup f(2) \rrbracket (A(1) \llbracket f(1) \mid cs \mid f(2) \rrbracket A(2))$.

Formally, we have the following definition of iterated parallel composition.

Definition B.25

$$\llbracket\, cs \,\rrbracket\; x : \{v_1, \ldots, v_n\} \bullet \llbracket\, ns(x) \,\rrbracket\; A(x) \mathrel{\widehat=} \begin{array}{l} A(v_1) \\ \llbracket ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, \ldots, v_n\} \bullet ns(x)\}\rrbracket \\ \left(\ldots \left(\begin{array}{l} A(v_{n-1}) \\ \llbracket ns(v_{n-1}) \mid cs \mid ns(v_n)\rrbracket \\ A(v_n) \end{array} \right) \right) \end{array}$$

Each step of the expansion takes a value v_i from the type of the indexing variable and creates a binary parallel composition. The left-hand side is the instantiation $A(v_i)$ with priority over the variables in $ns(v_i)$. The right-hand side is the expansion of the iterated parallel composition for the remaining values $\llbracket\, cs \,\rrbracket\; x : \{v_{i+1}, \ldots, v_n\} \bullet \llbracket\, ns(x) \,\rrbracket\; A(x)$; it has priority on all the variables that are in the set of the remaining variables (expressed as $\bigcup\{x : \{v_{i+1}, \ldots, v_n\} \bullet ns(x)\}$). Iterated interleaving is given a very similar definition that does not consider the synchronisation channel set.

3.1.2 Action Invocations, Parametrised Actions and Renaming

The semantics of a reference to an action name is given by the copy rule: it is the body of the action. Invocation of unnamed parametrised actions is defined simply as the substitution of argument for the formal parameter.

Definition B.29 $(x : T \bullet A)(e) \mathrel{\widehat=} A[e/x]$

The renaming of the local variables and state components is simply the syntactic substitution of the new names for the old ones.

3.1.3 Commands

The semantics of an assignment is rather simple: it never diverges and terminates successfully leaving the trace unchanged; of course, it sets the final values of the variables in the left-hand side to their new corresponding values. The remaining variables, denoted in the definition below by u ($u = v \setminus \{x_1, \ldots, x_n\}$), are left unchanged.

Definition B.31

$$\begin{array}{l} x_1, \ldots, x_n := e_1, \ldots, e_n \mathrel{\widehat=} \\ \qquad \mathbf{R}(\mathit{true} \vdash \mathit{tr}' = \mathit{tr} \land \neg\, \mathit{wait}' \land x_1' = e_1 \land \ldots \land x_n' = e_n \land u' = u) \end{array}$$

Specification statements only terminate successfully establishing the postcondition if its precondition holds; only the variables in the frame can be changed. Furthermore, on successful termination, the trace is left unchanged. In the definition below, we use u to denote the variables that are not in the frame ($u = v \setminus w$).

Definition B.32 $w : [\, pre, post \,] \mathrel{\widehat=} \mathbf{R}(pre \vdash post \land \neg\, wait' \land tr' = tr \land u' = u)$

Assumptions $\{g\}$ and coercions $[\, g\,]$ are simply syntactic sugaring for the specification statements $: [g, true]$ and $: [true, g]$, respectively.

Alternation can only diverge if none of the guards is $true$, or if any action guarded by a valid guard diverges; any of the guarded actions whose guard is valid can be chosen for execution.

Definition B.35

$$\mathbf{if} \parallel i \bullet g_i \rightarrow A_i \,\mathbf{fi} \mathrel{\widehat=} \mathbf{R}((\bigvee i \bullet g_i) \land (\bigwedge i \bullet g_i \Rightarrow \neg\, A_{i}{}^{f}_{f}) \vdash \bigvee i \bullet (g_i \land A_{i}{}^{t}_{f}))$$

The last command, variable block, is defined in terms of the UTP constructors **var** and **end**; the former begins the scope of a variable, and the latter ends it.

Definition B.36 $\mathbf{var}\ x : T \bullet A \mathrel{\widehat=} \mathbf{var}\ x : T;\ A;\ \mathbf{end}\ x : T$

In fact, as we discuss in Section 3.2, these are defined in the UTP as existential quantification on the dashed and undashed variables, respectively. As a consequence, we have the following corollary.

Corollary 3.1 $\mathbf{var}\ x : T \bullet A = \exists\, x, x' : T \bullet A$

The declaration of a variable x actually introduces both x and x' into scope.

Parametrisation by value, result, or by value-result, like those presented in [22], can be defined in terms of other existing *Circus* constructs, namely, variable blocks and assignments. For instance, in a parametrisation by value, the formal parameter receives the value of the actual argument, which is actually to be used by the action. Therefore, we may define it as follows.

Definition B.37 $(\mathbf{val}\ x : T \bullet A)(e) \mathrel{\widehat=} (\mathbf{var}\ x : T \bullet x := e;\ A)$, *provided* $x \notin FV(e)$.

Similar syntactic transformations can be applied to the other kinds of parameters.

3.1.4 Schema Expressions

Our semantics for schema expressions differs from the one presented in [105]. As previously discussed, Cavalcanti and Woodcock's models for *Circus* programs are Z specifications; hence, the semantics of a schema expression was simply the schema expression itself with some adjustments to take the UTP observational variables into account. We use the basic conversion rule of [22] to transform schema expressions into specification statements.

We assume that the schema expressions of the specification have already been normalised using the normalisation techniques presented in [107]. Besides, in *Circus*, the Z notations for input (?) and output (!) variables are syntactic sugar for undashed and dashed variables, respectively. This implies that we actually have a schema containing the declaration of dashed ($ddecl'$) and undashed ($udecl$) variables and the predicate that determines the effect of the schema. As a small abuse of notation, we use $ddecl$ also to stand for a comma-separated list of undashed variables introduced as dashed variables in $ddecl'$.

Definition B.40 $[udecl;\ ddecl' \mid pred] \mathrel{\widehat=} ddecl : [\exists\, ddecl' \bullet pred, pred]$

By way of illustration, let us consider a process with state $S \mathrel{\widehat=} [x : \mathbb{N} \mid x < 10]$. The semantics of the schema operation $Odd \mathrel{\widehat=} [\Delta S \mid x' \mathsf{\ mod\ } 2 \neq 0]$, which chooses any odd natural number below 10 for x, is as follows.

$$
\begin{array}{lr}
[\Delta S \mid x' \mathsf{\ mod\ } 2 \neq 0] & [\text{Normalisation}] \\
= [x, x' : \mathbb{Z} \mid x \in \mathbb{N} \wedge x' \in \mathbb{N} \wedge x' \mathsf{\ mod\ } 2 \neq 0 \wedge x < 10 \wedge x' < 10] & [\text{Definition B.40}] \\
= x : \left[\begin{array}{rcl} \exists\, x' : \mathbb{Z} \bullet x \in \mathbb{N} \wedge x' \in \mathbb{N} & & x \in \mathbb{N} \wedge x' \in \mathbb{N} \\ \wedge\, x' \mathsf{\ mod\ } 2 \neq 0 & , & \wedge\, x' \mathsf{\ mod\ } 2 \neq 0 \\ \wedge\, x < 10 \wedge x' < 10 & & \wedge\, x < 10 \wedge x' < 10 \end{array} \right] &
\end{array}
$$

First, we normalise the schema expression, and finally, we apply the Definition B.40. This specification statement has the expected behaviour: if the precondition of the schema operation is satisfied, then it chooses an odd natural number below 10 for x'; however, if the precondition is false, it aborts. The reactive behaviour of the schema is embedded in the semantics of specification statements (see Definition B.32 above).

3.1.5 *Circus* Processes

An explicitly defined process has an encapsulated state, a sequence $PPars$ of *Circus* paragraphs, and a main action A, which defines its behaviour. It declares the state components using a *Circus* variable block and behaves like A.

Definition B.41 $\mathbf{begin\ state}\ [decl \mid pred]\ PPars\ \bullet A\ \mathbf{end} \mathrel{\widehat=} \mathbf{var}\ decl \bullet A$

All the compound processes can be defined in terms of an explicit process specification. For instance, sequence, external and internal choice can be defined as follows.

Definition B.42 For $op \in \{\, ; \,, \Box \,, \sqcap \}$:

$$
\begin{array}{ll}
P\ op\ Q \mathrel{\widehat=} & \mathbf{begin\ state}\ State \mathrel{\widehat=} P.State \wedge Q.State \\
& \qquad P.PPar \wedge_{\Xi} Q.State \\
& \qquad Q.PPar \wedge_{\Xi} P.State \\
& \qquad \bullet\ P.Act\ op\ Q.Act \\
& \mathbf{end}
\end{array}
$$

The state of the process $P\ op\ Q$ is defined as the conjunction of the individual state of both P and Q; for simplicity, we assume that name clashes are avoided through renaming. Furthermore, every schema in the paragraphs of P (Q), specify an operation on $P.State$ ($Q.State$); they are not by themselves operations on $P\ op\ Q$. For this reason, we need to lift them to operate on the global *State*. For a sequence of process paragraphs $P.PPar$, the operation $P.PPar \wedge_{\Xi} Q.State$ stands for the conjunction of each schema expression in the paragraphs $P.PPar$ with $\Xi Q.State$; this indicates that they do not change the components of the state of process Q ($Q.State$). The main actions are composed in the same way using *op*; all the references from $P.Act$ to the components of $P.State$ are through schemas, which have already been conjoined with $\Xi Q.State$; the same comment applies to $Q.Act$.

For parallel composition and interleaving the only difference is that we must determine the state partitions of the operators. These are trivially the state components of each individual process as presented below.

Definition B.43

$$\begin{array}{ll} P \llbracket cs \rrbracket Q \mathrel{\widehat=} & \textbf{begin state } State \mathrel{\widehat=} P.State \wedge Q.State \\ & \quad P.PPar \wedge_{\Xi} Q.State \\ & \quad Q.PPar \wedge_{\Xi} P.State \\ & \quad \bullet P.Act \llbracket \alpha(P.State) \mid cs \mid \alpha(Q.State) \rrbracket Q.Act \\ & \textbf{end} \end{array}$$

The similar definition for interleaving is omitted here.

The semantics of hiding is very simple: all the process paragraphs are included as they are; the only change is in the main action, which we modify to include the hiding.

Definition B.45 $P \setminus cs \mathrel{\widehat=} \textbf{state } State \mathrel{\widehat=} P.State\ P.PPar\ \bullet P.Act \setminus cs\ \textbf{end}$

Our semantics for an indexed process $x : T \odot P$ is that of a parametrised process $x : T \bullet P$. However, all the communications within the corresponding parametrised processes are changed. For every channel c used in P, we have a freshly named channel c_i, which communicates pairs of values: the first element, the index, is a value i of type T, and the second element is the value of the original type of the channel. The semantics of the corresponding parametrised process is given using an extended channel environment δ that includes the new implicitly declared channels c_i.

Definition B.46 $x : T \odot P \mathrel{\widehat=} (x : T \bullet P)[c : usedC(P) \bullet c\_x.x]$

The notation $P[c : usedC(P) \bullet c\_x.x]$ denotes the change, in P, of all the references to every used channel c by a reference to $c\_x.x$. Since our semantics for indexed processes are parametrised processes, the semantics for their instantiation is simply a parametrised process invocation.

Some of the semantics for processes take the same approach that was taken for actions. For instance, the semantics of a reference to a process is the body of the process.

	Formal Representation	Description
C1	$\mathbf{C1}(P) \mathrel{\widehat{=}} P; Skip$	The value of the variable ref' has no relevance after termination
C2	$\mathbf{C2}(P) \mathrel{\widehat{=}} A \lbrack\!\lbrack ns_1 \mid ns_2 \rbrack\!\rbrack Skip$	A deadlocked process that refuses some events offered by its environment will still be deadlocked in an environment which offers even fewer events
C3	$\mathbf{C3}(P) \mathrel{\widehat{=}} \mathbf{R}(\neg A^f_f; true \vdash A^t_f)$	The precondition of a *Circus* process expressed as a reactive design contains no dashed variables

Table 3.3: Healthiness Conditions — *Circus* Processes

3.1.6 *Circus* Healthiness Conditions

From Theorems 3.1 and 3.3, we already know that every *Circus* action is **R** and **CSP1**-**CSP3** healthy. However, processes that can be defined using the notation of CSP also satisfy two other healthiness conditions: the value of ref' has no relevance after termination of **CSP4** processes and a deadlocked **CSP5** process that refuses some events offered by its environment will still be deadlocked in an environment that offers even fewer events. Both, **CSP4** and **CSP5**, are expressed in terms of CSP constructs that have a slightly different definition in *Circus*: **CSP4** processes satisfy the right unit law ($P; SKIP = P$) and **CSP5** processes satisfy the unit law of interleaving ($P \mathbin{|||} SKIP = P$) [54]. The healthiness conditions **C1** and **C2** presented in Table 3.3 lift these two healthiness conditions to state-rich *Circus* processes.

The last of the *Circus* healthiness conditions, **C3**, guarantees that every *Circus* action, when expressed as a reactive design, has no dashed variables in the precondition. Since *Circus* actions are **CSP1**-**CSP2** healthy, we use Theorem 3.2 to transform them into reactive designs; if they are originally already expressed so, this transformation has no effect whatsoever. The sequential composition of the precondition with *true* guarantees that only those actions with no dashed variables in the precondition will be a fixed-point of the function **C3**.

The last theorem regarding healthiness conditions guarantees that every *Circus* operator is indeed **C1**-**C3** healthy.

Theorem 3.6 *Every Circus action is **C1**, **C2**, and **C3** healthy.*

As for the similar theorems for **R** and **CSP**, the proof of this theorem is done by induction on the language; it is omitted here for the sake of conciseness.

3.2 Towards a Theorem Prover for *Circus*

In this section we present the details of the mechanisation of *Circus* and its theoretical basis [77], the UTP. For the sake of presentation, we do not present the Z generated by the ProofPower-Z document preparation tool, which has an awkward indentation for expressions. Instead, we present a better indented copy of the pretty-printed ProofPower-Z expressions. First, we introduce ProofPower-Z, a theorem prover that supports specifications and proofs in Z. Then, we present the mechanisation of the UTP theories of alphabetised relations, designs, reactive processes, and CSP processes. An account of how this mechanisation is done, and more interestingly, of what issues were raised and of our decisions, is presented here. This work provides tool support for further explorations of Hoare & He's unification, and for the mechanisation of languages based on this unification. More specifically, this work supports the mechanisation of *Circus*, whose description concludes this section. A summary of the material in this section is published in [77].

3.2.1 ProofPower-Z

ProofPower-Z is a higher-order tactic-based theorem prover implemented using New Jersey SML that supports specifications and proofs in Z. It extends ProofPower-HOL, which builds on ideas arising from research at the Universities of Cambridge [49] and Edinburgh [48]. Some of the extensions provided by the New Jersey SML were used in ProofPower-Z in order to achieve features such as a theory hierarchy, extension of the character set accepted by the metalanguage ML, and facilities for quotation of object language (Z or HOL) expressions, and for automatic pretty-printing of the representation of such expressions.

As it is an extension of ProofPower-HOL, definitions can be made using Z, HOL, and even SML, which is the input command language. ProofPower-Z also offers the possibility of simply defining a wide range of proof tactics, as opposed to Z/EVES, which can be used to reduce, and modularise proofs. Among other analysis support, ProofPower-Z provides syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. Using the subgoal package, goals can be split into simpler subgoals, and proved; proofs are finished once all the subgoals have been proved. This allows users to focus their attention on a particular part of the theorem at each time. The Z notation used in ProofPower-Z is almost the same as that of the Z standard. Those points where it differs from the standard, and which are relevant in this section, are pointed out as needed.

ProofPower-Z comes with a large number of formally verified theories, including elementary number theory, algebra, set theory, linear arithmetics, and many Z related theories are also included. Furthermore, as we intend to mechanise the UTP model of relations, ProofPower-Z was a more convenient choice because it is based on sets, rather than functions, like PVS [2]. Our project is largely funded by QinetiQ: they intend to use *Circus* in their development process. ProofPower-Z is the theorem prover that they use routinely and was a pragmatic choice as a basis to provide a theorem prover for *Circus*.

3.2.2 Design Issues

This section describes the issues raised during the automation of the UTP. The first difficulty that we faced was that the name of a variable is used to refer both to the name itself and to its value. For instance, in the relation $(\{x\}, x = 0)$, the left-most x indicates that x is the name of a variable in the alphabet, while the right-most x stands for the value of x. We make explicit the difference between a variable name and a variable value.

We discarded the option of giving an axiomatic semantics to relations, since we would not be able to use most of the theorems that are built-in in ProofPower-Z to reason about sets and other models. Our relations are pairs of sets.

Since we want to prove refinement laws, our mechanisation gives the possibility of expressing and proving meta-theorems. A shallow embedding, in which the mapping from language constructs to their semantic representation is part of the meta-language, would not allow us to express such theorems. We use a deep embedding, where the syntax and the semantics of alphabetised relations is formalised inside the host language. The deep embedding has the additional advantage of providing the possibility of introducing new predicate combinators.

The syntax of relations and designs could be expressed as a data type (Z free types), say $REL\_PREDICATE$, for the relations. In this case, the semantics would be given as a partial ($\nrightarrow$) function $f : REL\_PREDICATE \nrightarrow REL\_PREDICATE$. If we took this approach, most of the proofs would be by induction over $REL\_PREDICATE$. Any extension to the language would require proving most of the laws again. Instead, we express the language constructors as functions; this is a standard approach in functional languages. Extensions require only the definition of the new constructors, and that they preserve any healthiness conditions; no proofs need to be redone.

Using SML as a meta-language would not give us a deep embedding. We were left with the choice of Z or HOL. If we used HOL as meta-language, reusing the definitions of Z constructs would not be possible, because they are written in SML. Because of our knowledge of Z, and the expressiveness of its toolkit, we have used Z as our meta and target language.

In Figure 3.1, we present our hierarchy of theories. In order to handle sequences, we extend ProofPower-Z's theory *z-library*; the result is *utp-z-library*. The theory *utp-rel* is that of general UTP relations. It includes basic alphabetised operators like conjunction and existential quantification; relational operators like alphabet extension, sequential composition, and skip; and refinement. Like all our theories, it includes the operator definitions and their laws.

Two theories inherit from *utp-rel*: *utp-okay* is concerned with the observational variable *okay*, and *utp-wtr* with *wait*, *trace*, and *ref*. These are the main variables of the theory of reactive processes. The theory *utp-okay* is the parent of *utp-des*, the theory for designs. Along with *utp-wtr*, *utp-okay* is also the parent of the reactive processes theory (*utp-rea*), which redefines part of *utp-rel*. The theory for CSP processes, *utp-csp*, inherits from both *utp-rea* and *utp-des*. The theory for *Circus* (*utp-circus*) inherits from *utp-csp*. Our proofs of the laws of a theory does not expand definitions of its parent theory; it uses the parent's laws. This provides modularisation and encapsulation.

3.2.3 Relations

A name is an element of the given set $[NAME]$. Each relation has an alphabet of type

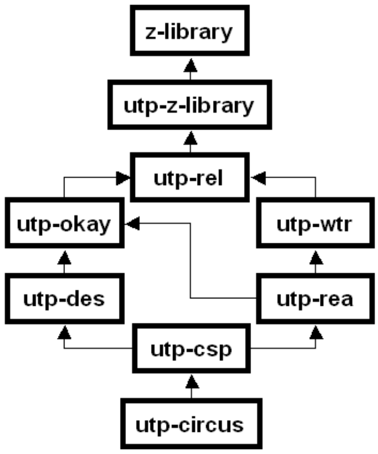


Figure 3.1: Theories in the UTP

$ALPHABET \mathrel{\widehat=} \mathbb{P}\, NAME$. The Z abbreviation $N == A$ is provided as $N \mathrel{\widehat=} A$ in ProofPower-Z; it gives a name N to the mathematical object A. Every alphabet a contains an input alphabet of undashed names, and an output alphabet of dashed names. Instead of using free types, which would lead to more complicated proofs in ProofPower-Z, we use the injective ($\rightarrowtail$) function $dash : NAME \rightarrowtail NAME$ to model name decoration. The set of *dashed* names is defined as the range of *dash*. The complement of this set is the set of *undashed* names; hence, names are either *dashed* or *undashed*, but multiple dashes are allowed.

For the sake of conciseness, we omit the definitions of the functions in_a and out_a, which return the input and the output alphabets of a given alphabet. All the definitions and proof scripts can be found in [71].

An alphabet a in which $n \in a \Leftrightarrow n' \in a$, for every *undashed* name n, is called *homogeneous*. For us, n' is mechanised as $dash\, n$. Similarly, a pair of alphabets $(a1, a2)$ is *composable* if $n \in a2 \Leftrightarrow n' \in a1$, for every *undashed* name n.

A value is an element of the free-type $VALUE$, which can be an integer, a boolean, a channel, a sequence of values, a set of values, a pair of values, or a special synchronisation value.

$$\begin{aligned} VALUE ::= {} & Int(\mathbb{Z}) \mid Bool(BOOL) \mid Channel(NAME) \mid Seq(\mathrm{seq}\ VALUE) \\ \mid {} & Set(\mathbb{P}\, VALUE) \mid Pair(VALUE \times VALUE) \mid Sync \end{aligned}$$

In ProofPower-Z, $Bool(BOOL)$ stands for the Z constructor $Bool\langle\!\langle BOOL \rangle\!\rangle$, which introduces a collection of constants, one for each element of the set $BOOL$. The ProofPower-Z type $BOOL$ is the booleans. The type $VALUE$ can be extended without any impact on the proofs because they do not depend on its structure.

Although we are defining an untyped theory, the observational variables have types; for instance, *okay* is a boolean. For this reason, we specify some types; for instance, booleans are in the set $BOOL\_VAL \mathrel{\widehat=} \{Bool(true), Bool(false)\}$, channels are in the set

$CHANNEL\_VAL \mathrel{\widehat=} \{n : NAME \bullet Channel(n)\}$, and events are in the set of events $EVENT\_VAL \mathrel{\widehat=} \{c : CHANNEL\_VAL;\ v : VALUE \bullet Pair(c, v)\}$.

Three definitions allow us to abstract from the syntax of expressions. The set of relations between values is $RELATION \mathrel{\widehat=} VALUE \leftrightarrow VALUE$. The set of unary functions is $UNARY\_F \mathrel{\widehat=} VALUE \mathrel{+\!\!\!\rightarrow} VALUE$; similarly, for binary functions we have the set $BINARY\_F \mathrel{\widehat=} (VALUE \times VALUE) \mathrel{+\!\!\!\rightarrow} VALUE$, which defines the set of partial functions from pairs of values to values. For instance, the sum function is $\{(Int(0), Int(0)) \mapsto Int(0), (Int(0), Int(1)) \mapsto Int(1), \ldots\}$. An expression can be a value, a name, a relation, or a unary or binary function application.

$$
\begin{aligned}
EXPRESSION ::= {} & Val(VALUE) \mid Var(NAME) \\
\mid {} & Rel(RELATION \times EXPRESSION \times EXPRESSION) \\
\mid {} & Fun_1(UNARY\_F \times EXPRESSION) \\
\mid {} & Fun_2(BINARY\_F \times EXPRESSION \times EXPRESSION)
\end{aligned}
$$

The definitions for unary functions, binary functions, and relations only deal with values. For instance, for a given unary function f, the expression $Fun_1(f, e)$ can only be evaluated once e is evaluated to some $VALUE$.

A binding is defined as $BINDING \mathrel{\widehat=} NAME \mathrel{+\!\!\!\rightarrow} VALUE$, and $BINDINGS$ is the set of bindings. Given a binding b and an expression e with free-variables in the domain (dom) of b, $Eval(b, e)$ gives the value of e in b (beta-reduction).

A relation is modelled in our work by the type $REL\_PREDICATE$ defined below. A relation is a pair: the first element is its alphabet, and the second is a set of bindings, which gives us all bindings that satisfy the UTP predicate modelled by the relation. The domain of the bindings must be equal to the alphabet. Optional models in which this restriction could be relaxed are possible; however, they would lead us to more complex definitions, as we discuss in Section 3.3. The set comprehension $\{x : s \mid p \bullet e\}$ denotes the set of all expressions e, such that x is taken from s and satisfies the condition p. Usually, e contains one or more free occurrences of x. The *true* condition and the constructor e may be omitted.

$$
\begin{aligned}
& REL\_PREDICATE \mathrel{\widehat=} \\
& \quad \{a : ALPHABET;\ bs : BINDINGS \mid (\forall b : bs \bullet \operatorname{dom} b = a) \bullet (a, bs)\}
\end{aligned}
$$

This corresponds directly to the definition of alphabetised predicates of the UTP.

In our work, we use Z axiomatic definitions, which introduce constrained objects, to define our constructs. For instance, let us consider the following axiomatic definition.

$$
\begin{array}{|l}
x : s \\
\hline
p
\end{array}
$$

It introduces a new symbol x, an element of s, satisfying the predicate p.

Our first construct represents truth. For a given alphabet a, $True_R\ a$ is defined as the pair with alphabet a, and with all the bindings with domain a.

$$True_R : ALPHABET \rightarrow REL\_PREDICATE$$
$$\forall a : ALPHABET \bullet True_R\ a = (a, \{b : BINDING \mid \mathrm{dom}\ b = a\})$$

In our work, we subscript the names of the constructs in order to make it easier to identify to which theory they belong; we use R for the theory of relations.

Nothing satisfies *false*: the second element of $False_R\ a$ is the empty set.

$$False_R : ALPHABET \rightarrow REL\_PREDICATE$$
$$\forall a : ALPHABET \bullet False_R\ a = (a, \varnothing)$$

This operator is the main motivation for representing relations as pairs. If we had defined relations just as a set of bindings with the same domain a, which would be considered as the alphabet, we would not be able to tell the difference between $False_R\ a_1$ and $False_R\ a_2$, since both sets would be empty. Besides, it is important to notice the difference between $True_R\ \varnothing$ and $False_R\ \varnothing$: the former has a set that contains one empty set of bindings as its second element, and the latter has the empty set as its second element.

As we are working directly with the semantics of predicates, we are not able to give a syntactic characterisation of free variables. Instead, we have the concept of an unrestricted variable, which is actually not equivalent to that of a free-variables. As a matter of fact, if a variable is not a free-variable of a predicate, then it is unrestricted, but the reciprocal does not hold. For instance, x is an unrestricted variable in $x = x$, but it is a free-variable of this predicate.

$$UnrestVar : REL\_PREDICATE \rightarrow \mathbb{P}\ NAME$$
$$\forall u : REL\_PREDICATE \bullet$$
$$UnrestVar\ u = \{n : u.1 \mid \forall b : u.2;\ v : VALUE \bullet b \oplus \{n \mapsto v\} \in u.2\}$$

For a relation u, a name n from its alphabet is unrestricted if, for every binding b of u, all the bindings obtained by changing the value of n in b are in u. In Z, $f \oplus g$ stands for the relational overriding of f with g; furthermore, $t.n$ refers to the *n-th* element of a tuple t.

All usual predicate combinators are defined. Conjunctions and disjunctions extend the alphabet of each relation to the alphabet of the other. The function $\oplus_R$ is alphabet extension; the values of the new variables are left unconstrained. In the following definition we make use of the Z domain restriction $A \lhd B$: it restricts a relation $B : X \leftrightarrow Y$ to a set A, which must be a subset of X, ignoring any member of B whose first element is not a member of A.

$$\_ \oplus_R \_ : REL\_PREDICATE \times ALPHABET \rightarrow REL\_PREDICATE$$
$$\forall u : REL\_PREDICATE;\ a : ALPHABET$$
$$\bullet\ u \oplus_R a = (u.1 \cup a, \{b : BINDING \mid (u.1 \lhd b) \in u.2 \wedge \mathrm{dom}\ b = u.1 \cup a\})$$

The conjunction is defined as the union of the alphabets and the intersection of the extended set of bindings of each relation.

$$
\begin{array}{|l}
\_ \wedge_R \_ : REL\_PREDICATE \times REL\_PREDICATE \rightarrow REL\_PREDICATE \\
\hline
\forall\, u1, u2 : REL\_PREDICATE \bullet \\
\quad u1 \wedge_R u2 = (u1.1 \cup u2.1, (u1 \oplus_R u2.1).2 \cap (u2 \oplus_R u1.1).2)
\end{array}
$$

The definition of disjunction is similar, but the union of the extend set of bindings is the result. We have proved that these definitions are idempotent, commutative, and associative, and that they distribute over each other. We have also proved that $True_R$ is the zero for disjunction and the unit for conjunction; similar laws were also proved for $False_R$. However, restrictions on the alphabets must be taken into account. For example, we have the unit law for conjunction. The ProofPower-Z output notation $n \vdash t$ gives name n to a theorem t. Besides, in Z, the quantification $\forall\, x : a \mid p \bullet q$ corresponds to the predicate $\forall\, x : a \bullet p \Rightarrow q$.

$$
\begin{array}{l}
REL\_True\_ \wedge_R \_id\_thm1 \\
\vdash \forall\, a : ALPHABET;\ u : REL\_PREDICATE \mid a \subseteq u.1 \bullet u \wedge_R True_R\ a = u
\end{array}
$$

As expected, the conjunction of a relation u with $True_R$ is u, but the alphabet of $True_R$ must be a subset of the alphabet of u. Otherwise, the conjunction may have an alphabet other than that of u and the theorem does not hold.

The negation of a relation r does not change its alphabet. Only those bindings b that do not satisfy r ($b \notin r.2$) are included in the resulting bindings. For the sake of conciseness, we omit the definitions of implication ($\_ \Rightarrow_R \_$), equivalence ($\_ \Leftrightarrow_R \_$), and conditional ($\_ \lhd_R \_ \rhd_R \_$), which can be trivially defined in terms of the previously defined operators.

The function $-_R$ removes variables from the alphabet of a relation using domain anti-restriction (domain subtraction) to remove names from the set of bindings. It is defined as $u -_R a = (u.1 \setminus a, \{b : u.2 \bullet a \ntriangleleft b\})$. Complementary to domain restriction, the domain anti-restriction $A \ntriangleleft B$, ignores any member of B, whose first element is a member of A. Existential quantification $\exists_{-R}$ simply removes the quantified variables from the alphabet and changes the bindings accordingly.

$$
\begin{array}{|l}
\exists_{-R} : (ALPHABET \times REL\_PREDICATE) \rightarrow REL\_PREDICATE \\
\hline
\forall\, a : ALPHABET;\ u : REL\_PREDICATE \bullet \exists_{-R}(a, u) = u -_R a
\end{array}
$$

Universal quantification $\forall_{-R}(a, u)$ is defined as $\neg_R \exists_{-R}(a, \neg_R u)$.

In the definition of the CSP *SKIP*, Hoare and He use another existential quantification, in which the quantified variables are not removed from the alphabet. We define this new quantifier $\exists_R(a, u)$ as $(\exists_{-R}(a, u)) \oplus a$; we remove the quantified variables from the alphabet and include them again, leaving their values unrestricted.

Our sequential composition $u1; u2$ is not defined as in the UTP [54], an existential quantification on the intermediary state; the motivation is the simplification of our proofs. In the UTP definition [54], the existential quantification is described using new

0-subscripted names to represent the intermediate state. Its mechanisation requires two functions: one for creating new names, and another one for expressing substitution of names. Any proof on sequential composition would require induction on both functions.

Relations can only be combined in sequence if their alphabets are *composable*. If we defined sequential composition as a partial function, domain checks would be required during proofs. Instead, we define a total function on well-formed pairs of relations, WF_Semi_R, which have composable alphabets.

$$
\begin{array}{|l}
\_;_R\_ : WF\_Semi_R \rightarrow REL\_PREDICATE \\
\hline
\forall\, u1\_u2 : WF\_Semi_R \bullet \\
\quad u1\_u2.1 \,;_R\, u1\_u2.2 = \\
\quad\quad (in\_a\; u1\_u2.1.1 \cup out\_a\; u1\_u2.2.1, \\
\quad\quad \{b1 : u1\_u2.1.2;\; b2 : u1\_u2.2.2 \\
\quad\quad\; \mid (\forall\, n : \mathrm{dom}\, b2 \mid n \in undashed \bullet b2(n) = b1(dash\; n)) \\
\quad\quad\; \bullet (undashed \lhd b1) \cup (dashed \lhd b2)\})
\end{array}
$$

The alphabet of a sequential composition $u_1;_R\; u_2$ is composed of the input alphabet of the first relation and the output of the second relation. For each pair of bindings (b_1,b_2) from u_1 and u_2, respectively, we make a combination of all input values in b_1 (*undashed* names) with output values in b_2 (*dashed* names). However, only those pairs of bindings in which the final values of all names in b_1 correspond to their initial values in b_2 are taken into consideration in this combination.

The UTP defines an alphabet extension that enables sequential composition to be applied to operands with non-composable alphabets. The function $+_R$ differs from $\oplus_R$ in that it restricts the value of the new name. For a given predicate P and name n, it returns the predicate $P \wedge_R (n' =_{\{n',n\}} n)$.

In our work the skip is defined as the function defined below. Given a well-formed alphabet a, it does not change the alphabet and returns all the bindings b with domain a, in which for every *undashed* name n in a, $b\; n = b\; n'$. The type WF_Skip_R is the set of all *homogeneous* alphabets.

$$
\begin{array}{|l}
\Pi_R : WF\_Skip_R \rightarrow REL\_PREDICATE \\
\hline
\forall\, a : WF\_Skip_R \bullet \\
\quad \Pi_R\; a = (a, \{b : BINDING \\
\quad\quad\quad\quad \mid \mathrm{dom}\, b = a \\
\quad\quad\quad\quad\; \wedge (\forall\, n : a \mid n \in undashed \bullet b(n) = b(dash\; n))\})
\end{array}
$$

Other programming constructs like variable blocks and assignments are also included in this theory. For instance, we present below the definitions of variable declaration and *undeclaration*.

$$
\begin{array}{|l}
\mathbf{var}_R, \mathbf{end}_R : WF\_Var\_End_R \rightarrow REL\_PREDICATE \\
\hline
\forall\, a\_n : WF\_Var\_End_R \bullet \mathbf{var}_R\; a\_n = \exists\__R(\{a\_n.2\}, \Pi_R\; a\_n.1) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\; \mathbf{end}_R\; a\_n = \exists\__R(\{dash\; a\_n.2\}, \Pi_R\; a\_n.1)
\end{array}
$$

The type $WF\_Var\_End_R$ is the set of pairs (a, n), such that a is an *homogeneous* alphabet that contains both n, which must be an *undashed* name, and n'. Further definitions can also be found in [71].

We now turn to the definition of refinement as the universal implication of relations. The universal closure used in UTP [54] is defined $\langle_R\, u\,\rangle_R = \forall_{-R}(u.1, u)$. We have used angled brackets, instead of the square brackets of [54], because of problems with the LaTeX automatically generated by the ProofPower's document preparation tool. For a pair of relations (u_1,u_2), such that $(u_1, u_2) \in WF\_REL\_PREDICATE\_PAIR$ (both have the same alphabet), we have that u_1 is refined by u_2, if, and only if, for all names in their alphabets, $u_2 \Rightarrow u_1$. This is expressed by the definition below.

$$
\begin{array}{|l}
\_ \sqsubseteq_R \_ : WF\_REL\_PREDICATE\_PAIR \rightarrow REL\_PREDICATE \\
\hline
\forall\, u1\_u2 : WF\_REL\_PREDICATE\_PAIR \bullet \\
\qquad u1\_u2.1 \sqsubseteq_R u1\_u2.2 = \langle_R\, (u1\_u2.2 \Rightarrow_R u1\_u2.1)\,\rangle_R
\end{array}
$$

We have proved that our interpretation of refinement is, as expected, a partial order [71]. Moreover, the set of relations with alphabet a is a complete lattice.

Only functions $f : REL\_PREDICATE \nrightarrow REL\_PREDICATE$ whose domain is a set of relations with the same alphabet are considered in the theory of fixed-points. We call the set of such functions $REL\_FUNCTION$. The definition of the weakest fixed-point of a function $f : REL\_FUNCTION$ is standard. The greatest fixed-point is defined as the least upper bound of the set $\{X \mid X \sqsubseteq f(x)\}$.

3.2.4 Proving Theorems

We have built a theory with more than two-hundred and seventy laws on alphabets, bindings, relational predicates, and laws from the predicate calculus. In what follows, we illustrate our approach in their proofs and the use of the facilities provided by ProofPower-Z.

The proof of one of our laws is shown in Figure 3.2: the weakest fixed-point law $\forall\, F, Y \bullet F(Y) \sqsubseteq Y \Rightarrow \mu\, F \sqsubseteq Y$. We set our goal to be the law we want to prove using the SML command *set_goal*. It receives a list of assumptions and the proof goal. In our case, since we are not dealing with standard predicates, we must explicitly say that relations are $True_R$.

We start our proof by rewriting the Z empty set definition (*rewrite_tac*) and stripping the left-hand side of the implication into the assumptions (*z_strip_tac*). The SML command a applies a tactic to the current goal; the tactical $REPEAT$ applies the given tactic as many times as possible. The next step is to rewrite the definition of least fixed-point in the conclusion: we use forward chaining in the assumptions (*all_asm_fc_tac*), giving our Z definition of least fixed-point as argument, and use the new assumption to rewrite the conclusion(*asm_rewrite_tac*).

The application of a previously proved theorem, *REL_lower_bound_thm*, concludes our proof. However, it requires some assumptions, before being applied. We introduce them in the assumption list using the tactic *lemma_tac*. The first condition is that Y is an element of the set of relations u, with an alphabet a, such that $F(u) \sqsubseteq_R u$. We

```
SML
set_goal([], ⌜∀ F : REL_FUNCTION;
      Y : REL_PREDICATE
      | Y ∈ dom F
            ∧ (F(Y) ⊑_R Y = True_R∅)
      • μ_R(F) ⊑_R Y = True_R∅ ⌝);
a (rewrite_tac[]);
a (REPEAT z_strip_tac);
a (all_asm_fc_tac[z_get_spec ⌜μ_R⌝]);
a (asm_rewrite_tac[]);
a ( (PC_T1 "initial"
      lemma_tac
      ⌜Y ∈ {u : REL_PREDICATE
      | a = u.1 ∧ F u ⊑_R u = True_R{}} ⌝)
   THEN1   (asm_prove_tac[]) );
a (all_asm_fc_tac[]);
```

```
SML
a ( (lemma_tac
      ⌜{u : REL_PREDICATE
        | a = u.1 ∧ F u ⊑_R u = True_R{}}
      ∈ ℙ REL_PREDICATE ⌝)
  THEN1 (PC_T1 "z_sets_ext" asm_prove_tac[]) );
a ( (lemma_tac
      ⌜(a, {u : REL_PREDICATE
            | a = u.1 ∧ F u ⊑_R u = True_R{}})
       ∈ WF_Glb_R_Lub_R ⌝)
   THEN1
   ((rewrite_tac[z_get_spec ⌜WF_Glb_R_Lub_R ⌝])
      THEN
      (PC_T1 "z_sets_ext" asm_prove_tac[])) );
a (apply_def REL_lower_bound_thm
      ⌜(a≘a, u≘Y,
        us≘{u : REL_PREDICATE
            | a = u.1 ∧ F u ⊑_R u = True_R{}}) ⌝);
```

Figure 3.2: Proof script for the weakest fixed-point theorem

use the tactical PC_T1 to stop ProofPower-Z from rewriting our expression by using the proof context *initial*, which is the most basic proof context. Furthermore, to avoid a new subgoal, we use the tactical $THEN1$ that applies the tactic in the right-hand side to the first subgoal generated by the tactic in the left-hand side. In our case, this proves that the assumption we are introducing is valid. The validity of the introduction of the first assumption is proved using *asm_prove_tac*, a powerful tactic that uses the assumptions in an automatic proof procedure. Next, after introducing the first condition explained above in the list of assumptions, we use forward chaining again to state the fact that the alphabet of Y is a.

The next step introduces the fact that the set to which Y belongs is in fact a set of *REL_PREDICATE*. The proof of the validity of this assumption uses ProofPower-Z's proof context *z_sets_ext*, an aggressive complete proof context for manipulating Z set expressions. The last assumption that is needed is the fact that the pair composed of the alphabet a and the set to which Y belongs, is indeed of type $WF\_Glb_R\_Lub_R$, which contains all sets of pairs (a, bs), in which every binding in the set bs has a as its alphabet. Its proof rewrites the conclusion using the Z definition of $WF\_Glb_R\_Lub_R$, and then uses the tactic *asm_prove_tac* in the *z_sets_ext* proof context. Finally, we use a tactic defined by us, *apply_def*, to instantiate the theorem *REL_lower_bound_thm* with the given values. The tactic *apply_def* instantiates the given theorem with the values given as arguments, and tries to rewrite the conclusion, using this instantiation.

ProofPower-Z has provided us with facilities that resulted in a rather short proof, for a quite complex theorem. Some of the facilities we highlight are forward chaining, use of existing and user-defined tactics, proof contexts, and automated proof tactics, such as *asm_rewrite_tac*.

3.2.5 *Okay* and Designs

The UTP theory of pre and postcondition pairs (designs) introduces an extra observational variable *okay*: it indicates that a program has started, and $okay'$ indicates that the program has terminated. In our theory *utp-okay*, we define *okay* as an *undashed* name ($okay : NAME \mid okay \in undashed$) ranging over the booleans. We restrict the type *BINDING* by determining that *okay* and $okay'$ are only associated with boolean values.

$$\forall\, b : BINDING \mid \{okay, dash\ okay\} \subseteq \mathrm{dom}\, b \bullet \\ \quad \{b\ okay, b(dash\ okay)\} \subseteq BOOL\_VAL$$

We could have introduced this restriction when we first defined *BINDING*, but as we intend to have modular independent theories, we postponed the restriction on observational variables used by specific theories.

Designs are defined in the theory *utp-des*. The set *ALPHABET_DES* is the set of all alphabets that contain *okay* and $okay'$. First we define *DES_PREDICATE*, the set of relations u, such that $u.1 \in ALPHABET\_DES$. Designs with precondition p and postcondition q are written $p \vdash q$ and defined as $okay \wedge p \Rightarrow okay' \wedge q$. The expression *okay* is the equality $okay =_a true$, which is mechanised in our work as $=_R (a, okay, Val(Bool(true)))$. For a given alphabet a, name n, and expression e, such that $n \in a$ and the free-variables of e are in a, the function $=_R (a, n, e)$ returns a relational predicate (a, bs), in which for every binding b in bs, $b\, n = Eval(b, e)$. A design is defined as follows.

$$\_ \vdash_D \_ : WF\_DES\_PREDICATE\_PAIR \rightarrow REL\_PREDICATE$$

$$\forall\, d : WF\_DES\_PREDICATE\_PAIR \bullet \\ \quad d.1 \vdash_D d.2 = (=_R (d.1.1, okay, Val(Bool(true))) \wedge_R d.1) \Rightarrow_R \\ \qquad (=_R (d.1.1, dash\ okay, Val(Bool(true))) \wedge_R d.2)$$

The members of *WF_DES_PREDICATE_PAIR* are pairs of relations (r_1, r_2) of the type *DES_PREDICATE* with the same alphabet. The turnstile is used by both ProofPower-Z and the UTP. The former uses it to give names to theorems, and the later uses it to define designs. In our work, we have kept both of them, but we subscript the UTP design turnstile with a D.

The most important result for designs, which is the motivation for its definition, has also been proved in our mechanisation: the left-zero law for $True_R$.

In this new setting, new definitions for Π_R and assignment are needed. The skip for designs Π_D is defined in terms of the relational skip Π_R as follows.

$$\Pi_D : WF\_Skip_D \rightarrow REL\_PREDICATE$$

$$\forall\, a : WF\_Skip_D \bullet \Pi_D\, a = True_R\, a \vdash_D (\Pi_R\, a)$$

The type WF_Skip_D is formed by all the *homogeneous* alphabets that contain *okay* and $okay'$. The new definition of assignment uses the relation assignment in a very similar way and is omitted here.

Designs are also characterised by two healthiness conditions. The first, **H1**, guarantees that observations cannot be made before the program starts. We define $H1(d) = okay \Rightarrow d$ as $H1(d) = (=_R (\{okay\}, okay, Val(Bool(true)))) \Rightarrow_R d$. The set of relations that satisfy a healthiness condition h is the set of relations r such that $h(r) = r$. For instance, $H1\_healthy = \{d : REL\_PREDICATE \mid H1(d) = d\}$.

An $H2\_healthy$ relation does not require non-termination. In previous research [31], Cavalcanti and Woodcock presented a way of expressing **H2** in terms of an idempotent function: $H2(P) = P;\ J$, where $J \mathrel{\widehat{=}} ((okay \Rightarrow okay') \wedge v' = v)$. We express $v' = v$ as the relational skip Π_R on the alphabet containing the names in the lists v and v'. We define J as a function that takes an alphabet a' containing only *dashed* variables, and yields the relation presented below, where $A = a \cup a'$, and a is obtained by undashing all the names in a'.

$$(okay =_A true \Rightarrow_R okay' =_A true) \wedge_R \Pi_R(A \setminus \{okay, okay'\})$$

Our definition of the function $H2$ is presented below.

$$\begin{array}{|l} H2 : REL\_PREDICATE \rightarrow\!\!\!\!\!\!+ \; REL\_PREDICATE \\ \hline \forall\, d : REL\_PREDICATE \mid dash\ okay \in d.1 \bullet H2\, d = (d;_R (J(out\_a\ d.1))) \end{array}$$

The function $H2$ is partial because J defines a relation that includes $okay$ and $okay'$ in its alphabet, and hence, the alphabet of a relation d that can be made $H2\_healthy$ must contain $okay'$ in order to be *composable* with $J\,(out\_a\ d.1)$. In order to reuse the results in [31], we use this definition for **H2**.

More than thirty laws from previous work [54, 31], involving design and their healthiness conditions, have been included in our theory of designs. Their proofs do not expand any definition in the relations theory. Many laws were included in the relations theory, in order to carry out proofs in the designs theory.

3.2.6 WTR and Reactive Processes

The behaviour of reactive processes cannot be expressed only in terms of their final states; interactions with the environment (events) need to be considered. Besides $okay$, in the theory of reactive processes we have the observational variables tr,$wait$, and ref. The variable $wait$ records whether the process has terminated or is interacting with the environment in an intermediate state. Since it is a boolean, the definition of $wait$ is similar to that of $okay$. The variable tr records the sequence of events in which the process has engaged; it has type SEQ_EVENT_VAL. The variable ref is a set of events in which the process may refuse to engage; its type is SET_EVENT_VAL. The definitions of these variables are in the theory *utp-wtr*. In the theory *utp-rea*, we define $REA\_PREDICATE$, the set of relations whose alphabet is a member of $ALPHABET\_REA$; this is the set of alphabets that contain $okay$, tr, $wait$, ref, and their *dashed* counterparts.

As with designs, healthiness conditions characterise the reactive processes. The first healthiness condition **R1** states that the history of interactions of a process cannot be changed, therefore, the value of tr can only get longer. Our definition uses a function $\leq_R$ (sequence prefixing), which is the Z prefixing relation lifted to $VALUE$s.

$$\begin{array}{|l} \_ \leq_R \_ : VALUE \leftrightarrow VALUE \\ \hline (\_ \leq_R \_) = \{s1, s2 : SEQ\_VAL \mid ((Seq^\sim)\, s1)\ prefix_Z\ ((Seq^\sim)\, s2)\} \end{array}$$

The type SEQ_VAL is defined as $\{s : \mathrm{seq}\, VALUE \mid Seq(s)\}$. The type $\mathrm{seq}\, VALUE$ is the set of all Z sequences of $VALUE$s; the application of Seq to a member s of this set gives the $VALUE$ that corresponds to s. The Z sequence prefixing $prefix_Z$ is defined in *utp-z-library* and $^\sim$ stands for the Z relational inverse operator.

The definition of $R1$ below mechanises the function $\mathbf{R1}(P) = P \wedge tr \leq tr'$.

$$
\begin{array}{|l}
R1 : REL\_PREDICATE \rightarrow REL\_PREDICATE \\
\hline
\forall\, r : REL\_PREDICATE \bullet \\
\quad R1\, r = r \wedge_R (=_{+R} (\{tr, dash\, tr\}, \\
\qquad\qquad\qquad\qquad Rel((\_ \leq_R \_), Var(tr), Var(dash\, tr)), \\
\qquad\qquad\qquad\qquad Val(Bool(true))))
\end{array}
$$

In order to transform the expression $tr \leq tr'$ into a relational predicate, we assert that the expression $Rel((\_ \leq_R \_), Var(tr), Var(dash\, tr))$ is equal to $Val(Bool(true))$. We adopt the same strategy to lift all needed Z relational operators ($\in, \notin, \subseteq, \ldots$) and functions (using Fun_1 and Fun_2) to relational predicates.

The second healthiness condition establishes that a reactive process should not rely on events that happened before it started. We mechanise the following formulation from [54].

$$\mathbf{R2}(P(tr, tr')) = P(\langle\rangle, tr' - tr)$$

This requires that P is not changed if tr is taken to be the empty sequence, and tr' is taken to be $tr' - tr$. The notation $P(\langle\rangle, tr' - tr)$ is implemented using substitution; $R2(P)$ is defined as $P[\langle\rangle/tr][tr' - tr/tr']$.

The final healthiness condition **R3** defines the behaviour of a process that is still waiting for another process to finish: it should not start. In UTP [54], **R3** is defined as $\mathbf{R3}(P) = \Pi_{rea} \lhd wait \rhd P$, and is mechanised in our work as follows.

$$
\begin{array}{|l}
R3 : REA\_PREDICATE \nrightarrow REA\_PREDICATE \\
\hline
\forall\, r : REA\_PREDICATE \mid r.1 \in WF\_Skip_{REA} \bullet \\
\quad R3\, r = (\Pi_{REA}\, r.1) \lhd_R (=_R (\{wait\}, wait, Val(Bool(true)))) \rhd_R r
\end{array}
$$

This definition of $R3$ uses a conditional and the reactive skip Π_{REA}. Conditionals are defined only if both branches have the same alphabet and Π_{REA} is only defined for *homogeneous* reactive alphabets (WF_Skip_{REA}). For this reason, our definition reveals that $R3$ is not a total function: it can only be applied to *homogeneous* reactive relations.

A reactive process is a relation with a reactive alphabet a, which is $R\_healthy$; the function R is defined as $R(r) = R1(R2(R3(r)))$. Based on these definitions, more than sixty laws are part of our theory of reactive processes. Among other properties, they prove that the healthiness conditions for reactive processes are idempotent and commutative, and the closure of some of the program operators with relation to the healthiness conditions. They also explore relations between healthiness conditions for reactive processes and designs.

3.2.7 CSP

Our mechanisation of the CSP theory is based on the work in [31]. Basically, CSP processes are reactive processes that satisfy two other healthiness conditions; they can all be expressed as reactive designs: the result of applying **R** to a design. The first healthiness condition states that the only guarantee in the case of divergence ($\neg\ okay$) is that the trace can only be extended. It is mechanised as $CSP1\, r \mathrel{\widehat{=}}\ r \lor (\neg\ okay \land tr \leq tr')$.

The second healthiness condition is a recast of **H2**, presented in Section 3.2.5, with an extended reactive alphabet. The mechanisation of **CSP2** in ProofPower-Z reveals, as it does for **H2**, that this function is not total: it is only applicable to relational predicates that contain $okay'$, tr', $wait'$, and ref' in their alphabet.

$$
\begin{array}{|l}
CSP2 : REL\_PREDICATE \nrightarrow REL\_PREDICATE \\
\hline
\forall\, r : REL\_PREDICATE \mid \{dash\ okay, dash\ tr, dash\ wait, dash\ ref\} \subseteq r.1 \\
\qquad\qquad\qquad\qquad \bullet\ CSP2\, r = r;_R J(out\_a\ r.1)
\end{array}
$$

A $CSP\_PROCESS$ is a $CSP1\_healthy$ and $CSP2\_healthy$ reactive process. These are the sets containing all the **CSP1** healthy and **CSP2** healthy processes, respectively.

The $SKIP$ process terminates immediately. The initial value of ref is irrelevant, and it is quantified in the definition of $SKIP$.

$$
\begin{array}{|l}
SKIP : CSP\_PROCESS \\
\hline
SKIP = R(\exists_R\, (\{ref\}, \Pi_{REA}\ ALPHABET\_CSP))
\end{array}
$$

The set $ALPHABET\_CSP$ is the alphabet that contains only $okay$, tr, $wait$, ref, and their *dashed* counterparts. The existential quantification does not remove ref from the alphabet, as opposed to that used in the definition, for instance, of variable blocks.

The mechanisation of $do_{\mathcal{C}}$, used in the definition of prefix, is not as straightforward as one might expect. We have already discussed the mechanisation of the conditional (and its condition $wait'$), and the equality $tr' = tr$, which expresses that the trace does not change. The former is mechanised as $\_ \lhd_R (=_R (a, dash\ wait, Val(Bool(true)))) \rhd_R \_$, and the latter as $=_R (a, dash\ tr, tr)$. The mechanisation of $ev \notin ref'$ and $tr' = tr \frown \langle ev \rangle$ are a little more complex, as we explain now.

An $EVENT\_VAL$, as previously discussed, is a pair containing the channel name and a value; however, in CSP, one might write $n.e \rightarrow SKIP$, where e is actually an expression. For this reason, our implementation of $do_{\mathcal{C}}$ presented below receives two arguments: the name n of the channel and the communicated $EXPRESSION$ e. We assume that the observational variables cannot be used in a CSP specification. The type VAR_NAME is the set of all names that are not an UTP observational variable.

In our implementation, we need to express an event itself as an expression; with this purpose, we define a function $MkPair$ that receives a pair of $VALUE$s (v_1, v_2) and returns the $VALUE$ $Pair(v_1, v_2)$. The expression that defines the event as an expression is $Fun_2(MkPair, Val(Channel(n)), e)$; its evaluation will give us a pair where the first element is $Channel(n)$ and the second element is the evaluation of e. In the left-hand side of the condition, we lift the set non-membership relation $\notin_R$ to $VALUE$s in the same way we did for $\leq_R$ (page 56). In the right-hand side though, we use yet another function, $MkSingleton$, which receives a value v and returns the singleton sequence value $Seq(\langle v \rangle)$. The expression $Fun_1(MkSingleton, Fun_2(MkPair, Val(Channel(n)), e))$ corresponds to the expression $\langle ev \rangle$, where ev is itself an event expression. Finally, the same strategy to lift Z relations is applied to lift the Z concatenation function; however, we do not need to assert that the expression is equal to $true$.

$$
\begin{array}{|l}
do\_C : (VAR\_NAME \times EXPRESSION) \rightarrow CSP_PROCESS \\
\hline
\forall\, n : VAR\_NAME;\ e : EXPRESSION \bullet \\
\quad do\_C(n, e) = (\, (=_R (ALPHABET\_CSP, dash\ tr, Var(tr))) \\
\qquad\qquad \wedge_R\ (=_{+R} (\{ref, dash\ ref\}, \\
\qquad\qquad\qquad Rel((\_ \notin_R \_), \\
\qquad\qquad\qquad\quad Fun_2(MkPair, Val(Channel(n)), e), \\
\qquad\qquad\qquad\quad Var(dash\ ref)), \\
\qquad\qquad\qquad Val(Bool(true))))\,) \\
\qquad\qquad \lhd_R (=_R (ALPHABET\_CSP, dash\ wait, Val(Bool(true)))) \rhd_R \\
\qquad\qquad (\,(=_R (ALPHABET\_CSP, \\
\qquad\qquad\quad dash\ tr, \\
\qquad\qquad\quad Fun_2((\_ \frown_R \_), \\
\qquad\qquad\qquad Var(tr), \\
\qquad\qquad\qquad Fun_1(MkSingleton, \\
\qquad\qquad\qquad\quad Fun_2(MkPair, Val(Channel(n)), e)))))\,)
\end{array}
$$

This function is used in the definition of CSP prefix as a reactive design; furthermore, it is also used in the mechanisation of prefix in the *Circus* theory, which is the subject of the next section.

3.2.8 *Circus*

Although the constructors of CSP do not contain state variables, the set of processes described by the theory of CSP in the previous section contains processes that might have state components. By definition, a $CSP\_PROCESS$ is a $CSP1\_healthy$ and $CSP2\_healthy$ reactive process; the only restriction on the alphabet is that it must contain the observational variables and their *dashed* counterparts in the alphabet. Therefore, for us, *Circus* actions are members of $CSP\_PROCESS$; there is no need to define a new set of predicates. The definitions of the theory of *Circus*, *utp-circus*, follow directly from the semantics presented in Section 3.1. Besides, none of the *Circus* operators that are defined syntactically (i.e, iterated operators) are part of our mechanisation; the mechanisation of these operators is left as future work. In what follows, we present some of the more interesting definitions and discuss important aspects that were raised during this mechanisation.

We start with the definition of $Stop$. For a given *homogeneous* alphabet a that contains $ALPHABET\_CSP$ (WF_Skip_C), $Stop$ is the reactive design with a *true* precondition, which we mechanise using the relational $True_R$, and with the conjunction

$tr' =_a tr \wedge_R wait'$ as its postcondition.

$$
\begin{array}{|l}
Stop : WF\_Skip_C \rightarrow CSP\_PROCESS \\
\hline
\forall\, a : WF\_Skip_C \bullet \\
\quad Stop\ a = R(\ True_R\ a \vdash_D ((=_R (a, dash\ tr, tr)) \\
\qquad\qquad\qquad\qquad\qquad \wedge_R (=_R (a, dash\ wait, Val(Bool(true)))))\ )
\end{array}
$$

The mechanisation of *Skip a* is similar; however, besides leaving the trace unchanged, its postcondition requires termination ($\neg\ wait'$) and leaves the state components unchanged as we present below.

$$
\begin{array}{|l}
Skip : WF\_Skip_C \rightarrow CSP\_PROCESS \\
\hline
\forall\, a : WF\_Skip_C \bullet \\
\quad Skip\ a = R(\ True_R\ a \vdash_D ((=_R (a, dash\ tr, tr)) \\
\qquad\qquad\qquad\qquad\qquad \wedge_R (=_R (a, dash\ wait, Val(Bool(false)))) \\
\qquad\qquad\qquad\qquad\qquad \wedge_R \Pi_R(a \setminus ALPHABET\_CSP))\ )
\end{array}
$$

By giving the expression $a \setminus ALPHABET\_CSP$ as argument to the relational skip, we keep all the variables in a that are not in $ALPHABET\_CSP$ unchanged. *Chaos* is simply mechanised as the reactive design $R(False_R\ a \vdash_D True_R\ a)$, and sequential composition is trivially defined in terms of the corresponding relational operator presented in Page 52; it is "redefined" in this theory just for uniformity.

Before presenting the mechanisation of the semantics of guarded actions, we present below four new functions. These functions mechanise the substitutions A^b_c used in Section 3.1; in order to make it more alike the textual notation, we use a prefix notation for them. For instance, $A\,\sigma_f\,\omega_f$ mechanises the predicate A^f_f.

$$
\begin{array}{|l}
\omega_f, \omega_t, \sigma_f, \sigma_t : CSP\_PROCESS \rightarrow CSP\_PROCESS \\
\hline
\forall\, c : CSP\_PROCESS \bullet c\,\sigma_f = /_R(c, Val(Bool(false)), dash\ okay) \\
\qquad\qquad\qquad\qquad \wedge\ c\,\sigma_t = /_R(c, Val(Bool(true)), dash\ okay) \\
\qquad\qquad\qquad\qquad \wedge\ c\,\omega_f = /_R(c, Val(Bool(false)), wait) \\
\qquad\qquad\qquad\qquad \wedge\ c\,\omega_t = /_R(c, Val(Bool(true)), wait)
\end{array}
$$

Another important definition is that of predicates that can be used in the syntax of *Circus* specifications, which cannot mention any of the UTP observational variables. In our model, they are represented by the type *CIRCUS_PREDICATE*, which contains all the relational predicates in which the observational variables are in the alphabet, but left unrestricted within their types. On the other hand, in the syntax of *Circus*, conditions are predicates that contain no *dashed* variables. The type *CIRCUS_CONDITION* contains all the relational predicates, whose alphabet contains the observational variables, but in which the *dashed* variables that are not observational are *unrestricted* and the values of the observational variables are left unrestricted within their types.

A guarded action is defined in terms of a $CIRCUS\_CONDITION$ and a *Circus* action.

$$
\begin{array}{l}
\_ \&_C \_ : (CIRCUS\_CONDITION \times CSP\_PROCESS) \rightarrow CSP\_PROCESS \\
\hline
\forall g : CIRCUS\_CONDITION;\ a : CSP\_PROCESS \bullet \\
g \&_C a = R(\,(g \Rightarrow_R \neg_R (a\, \sigma_f\, \omega_f)) \\
\qquad\qquad \vdash_D \\
\qquad\qquad ((g \wedge_R (a\, \sigma_t\, \omega_f)) \\
\qquad\qquad\ \vee_R (\neg_R\, g \wedge_R\ (=_R (a.1, dash\ tr, tr)) \\
\qquad\qquad\qquad\qquad \wedge_R\ (=_R (a.1, dash\ wait, Val(Bool(true))))))\,)
\end{array}
$$

This definition derives directly from the semantics given in Section 3.1, but uses the new notation used in the mechanisation for substitution. The definitions of external and internal choice are trivial; they are omitted here for the sake of conciseness, but can be found in [71]. We now turn our attention to the prefix operators.

Simple prefix has a very similar definition to the CSP one; however, since *Circus* processes have state, the postcondition must guarantee that it is left unchanged. Besides, instead of defining two different functions, one for simple prefix followed by *Skip*, and other for simple prefix followed by any other action, we define a single function as presented below. The lack of uniformity is motivated by the convenience of implementation in ProofPower-Z.

$$
\begin{array}{l}
\_ \rightarrow_{CSync} \_ : (VAR\_NAME \times CSP\_PROCESS) \rightarrow CSP\_PROCESS \\
\hline
\forall c : VAR\_NAME;\ a : CSP\_PROCESS \bullet \\
\qquad c \rightarrow_{CSync} a = R(\ True_R\ a.1 \vdash_D do\_C(c, Val(Sync)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge_R \Pi_R(a \setminus ALPHABET\_CSP)\,);\ A
\end{array}
$$

Since no value is being communicated, we use the special synchronisation value $Sync$ as an argument to the function do_C. Besides, as with the *Circus* $Skip$, we also use the relational skip to state that the state components are left unchanged. If any value is being communicated, we have yet another function $\_ \rightarrow_C \_$, which, besides the channel name and the action, also receives an expression e; the only change in its definition is that e, instead of $Val(Sync)$, is given as argument to do_C.

The mechanisation of variable blocks is trivially done in terms of the relational operations that can be used to introduce and remove a variable from scope. Variable declaration is used in the expected way in the mechanisation of the input prefix, which we omit here for the sake of conciseness. The definitions are in direct correspondence with those in Section 3.2.3.

We now turn our attention to parallel composition, which we have mechanised in terms of a number of functions that correspond to elements of the original semantics in Section 3.1. In what follows, we explain some of them in detail and describe the remaining ones; their definitions can also be found in [71].

The function $MTrPar$ (parallel trace merge) presented below mechanises the function $\|_{cs}$ presented in Section 3.1: it receives a pair of traces $Pair(tr_1, tr_2)$ and a set of events $Set(cs)$ and returns a set $Set(s)$ containing all the possible sequences of events $Seq(e)$, where e is in the set of combinations of tr_1 and tr_2 according to cs. It uses a function $\_ \llbracket_Z \_ \rrbracket_Z \_$, which is defined in our *utp-z-library* that does the corresponding action for Z sequences.

$$
\begin{array}{|l}
MTrPar : (PAIR\_SEQ\_EVENT\_VAL \times SET\_EVENT\_VAL) \rightarrow \\
\qquad\qquad SET\_SEQ\_EVENT\_VAL \\
\hline
\forall\, ps : PAIR\_SEQ\_EVENT\_VAL;\ cs : SET\_EVENT\_VAL \bullet \\
\quad MTrPar(ps, cs) = Set(\{\, e : ((Seq^{\sim})((Pair^{\sim})\, ps).1) \\
\qquad\qquad\qquad\qquad\qquad \llbracket_Z ((Set^{\sim})\, cs) \rrbracket_Z \\
\qquad\qquad\qquad\qquad\qquad ((Seq^{\sim})((Pair^{\sim})\, ps).2) \\
\qquad\qquad\qquad\qquad \bullet\ Seq(e)\,\})
\end{array}
$$

The function *MTrParPred* receives a set of events cs and returns the mechanisation of the predicate $tr' - tr \in (1.tr - tr \parallel_{cs} 2.tr - tr)$. Its mechanisation is rather long, but trivial. We mechanise the expression $1.tr$ and $2.tr$ as the application of two injective functions one, two: $NAME \rightarrowtail NAME$; the only restriction on these functions is that their ranges are disjoint. The expression $1.tr \upharpoonright cs = 2.tr \upharpoonright cs$ is mechanised as the invocation of the function $MSync\, cs$. Two predicates *BranchesWaiting* and *BranchesNotWaiting* are defined in order to make the final definition of the merge function more easily readable: the former mechanises the predicate $1.wait \lor 2.wait$ and the latter mechanises the predicate $\neg\, 1.wait \land \neg\, 2.wait$. Yet another function, which has a rather long but simple definition, is *MRefPar*: it receives a set of events cs and returns the mechanisation of the predicate $ref' \subseteq ((1.ref \cup 2.ref) \cap cs) \cup ((1.ref \cap 2.ref) \setminus cs)$. Finally, the recursive function *MSt* returns a predicate that corresponds to the state merge. It receives three sets of names: the set st corresponds to the state components, and the sets $ns1$ and $ns2$ correspond to the names in the left-hand side and right-hand side partitions of the parallel composition, respectively. By way of illustration, given a state $st = \{x, y, z\}$ and partitions $ns1 = \{x\}$ and $ns2 = \{y\}$, the call $MSt(st, ns1, ns2)$ returns the predicate $x' = 1.x \land y' = 2.y \land z' = z$. The conditional presented in the merge function $M_{\parallel_{cs}}$ (Page 38) is mechanised as follows.

$$
\begin{array}{|l}
MWtRefStPar : (SET\_EVENT\_VAL \times ALPHABET \times ALPHABET \\
\qquad\qquad\qquad \times\, ALPHABET) \rightarrow REL\_PREDICATE \\
\hline
\forall\, ns1, ns2, st : ALPHABET;\ cs : SET\_EVENT\_VAL \bullet \\
\quad MWtRefStPar(cs, st, ns1, ns2) = \\
\qquad BranchesWaiting \land_R MRefPar(cs) \\
\qquad \lhd_R (=_R (dash_w ait, dash\ wait, Val(Bool(true)))) \rhd_R \\
\qquad BranchesNotWaiting \land_R MSt(st, ns1, ns2)
\end{array}
$$

It receives the synchronisation channel set cs, the set of names st of the state components, and the sets of names that correspond to the partitions $ns1$ and $ns2$. If the parallel combination is still waiting, then at least one of the branches is still waiting (*BranchesWaiting*), and the refusal set is defined by the function *MRefPar*; otherwise, both branches have terminated (*BranchesNotWaiting*) and the state is merged accordingly (*MSt*).

The merge function $M_{\|_{cs}}$ is mechanised as follows.

$$
\begin{array}{|l}
MPar : (SET\_EVENT\_VAL \times ALPHABET \times ALPHABET \\
\qquad\qquad \times ALPHABET) \rightarrow REL\_PREDICATE \\
\hline
\forall ns1, ns2, st : ALPHABET;\ cs : SET\_EVENT\_VAL \bullet \\
\qquad MPar(cs, st, ns1, ns2) = MTrParPred(cs) \wedge_R MSync(cs) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge_R MWtRefStPar(cs, st, ns1, ns2)
\end{array}
$$

It receives the same arguments as the function $MWtRefStPar$ presented above and returns the conjunction of the trace merge ($MTrParPred$), the predicate $MSync(cs)$ and the conditional described above.

Two more functions are needed in the mechanisation of parallel composition. Given two processes $a1$ and $a2$ and a synchronisation channel set cs, the first function, $DivPar$, returns the predicate that describes the condition on which $a1$ may diverge. Basically, it mechanises the predicate P1 presented in page 37.

$$
\begin{array}{|l}
DivPar : (CSP\_PROCESS \times CSP\_PROCESS \times SET\_EVENT\_VAL) \rightarrow \\
\qquad\qquad REL\_PREDICATE \\
\hline
\forall a1, a2 : CSP\_PROCESS;\ cs : SET\_EVENT\_VAL \bullet \\
\qquad DivPar(a1, a2, cs) = \exists_R(\{dash(one\ tr), dash(two\ tr)\}, \\
\qquad\qquad\qquad\qquad ((a1\, \sigma_f\, \omega_f);_C(=_R (a1.1, dash(one\ tr), Var(tr)))) \\
\qquad\qquad\qquad\qquad \wedge_R ((a2\, \omega_f);_C(=_R (a2.1, dash(two\ tr), Var(tr)))) \\
\qquad\qquad\qquad\qquad \wedge_R (MSync(cs)))
\end{array}
$$

As discussed in Section 3.1, in the parallel composition, we run both actions independently and merge their results afterwards. With this purpose, we use relabelling to capture their independent behaviours. In our work, the relabelling is done by the function U presented below.

$$
\begin{array}{|l}
U : ((NAME \rightarrowtail NAME) \times ALPHABET) \twoheadrightarrow REL\_PREDICATE \\
\hline
\forall f : (NAME \rightarrowtail NAME);\ a' : ALPHABET \mid a' \subseteq dashed \\
\quad \bullet (\exists a : ALPHABET \\
\qquad \mid a \subseteq undashed \wedge a' = dash (\!|\ a\ |\!) \\
\qquad \bullet U(f, a') = \\
\qquad\qquad (\{n : NAME \mid n \in a \bullet dash(f\ n)\} \cup a, \\
\qquad\qquad \{b : BINDING \mid \mathrm{dom}\, b = \{n : NAME \mid n \in a \bullet dash(f\ n)\} \cup a \\
\qquad\qquad\qquad \wedge (\forall n : NAME \mid n \in a \bullet b(dash(f\ n)) = b(n))\}))
\end{array}
$$

It receives a renaming function f (i.e, *one* and *two*) and an alphabet a' containing only *dashed* names, and returns a relational predicate whose alphabet is that resulting from the union of a' with the corresponding *undashed* alphabet a. We use the Z relational image to retrieve the *undashed* version of a'; for a given relation $D : X \leftrightarrow Y$, and a subset A of X, $D (\!|\ A\ |\!)$ returns the set of all elements in Y to which some element of A is related via D. The bindings of the resulting relational predicate are those whose domain is the same as the relation's alphabet, and in which the values of the final (*dashed*) values of the relabelled names $b(dash(f\ n))$ are the same as the value of their corresponding *undashed* original names.

Finally, the function that mechanises the parallel composition receives two process $a1$ and $a2$ with the same alphabet, the two partitions $ns1$ and $ns2$, which must be disjoint and contain only *undashed* names, and the synchronisation channel set cs. As described in Section 3.1, the parallel composition diverges if it is possible for either of the actions to diverge; this is expressed in the precondition of the resulting reactive design by using the function $DivPar$ as follows.

$$
\begin{array}{|l}
\_ \llbracket_C \_ \rrbracket_C \_ : (CSP\_PROCESS \times \\
\qquad\qquad\qquad (ALPHABET \times SET\_EVENT\_VAL \times ALPHABET) \times \\
\qquad\qquad\qquad CSP\_PROCESS) \rightarrow CSP\_PROCESS \\
\hline
\forall\, a1, a2 : CSP\_PROCESS;\ cs : SET\_EVENT\_VAL;\ ns1, ns2 : ALPHABET \bullet \\
\qquad a1 \llbracket_C (ns1, cs, ns2) \rrbracket_C\ a2 = \\
\qquad\qquad R(\,(\neg_R (DivPar(a1, a2, cs)) \wedge_R \neg_R (DivPar(a2, a1, cs))) \\
\qquad\qquad\quad \vdash_D \\
\qquad\qquad\quad (((\,(((a1\, \sigma_t\, \omega_f);_C U(one, out\_a\ a1.1)) \\
\qquad\qquad\qquad ((a2\, \sigma_t\, \omega_f);_C U(two, out\_a\ a2.1))) \\
\qquad\qquad\qquad +_R(\{tr\} \cup (a1.1 \setminus (ALPHABET\_CSP \cup dashed)))\,)\,;_C \\
\qquad\qquad\quad\ (MPar(cs, a1.1 \setminus (ALPHABET\_CSP \cup dashed), ns1, ns2)))\,)
\end{array}
$$

We use the function U to relabel the final values of the execution of actions $a1$ and $a2$; the relabelling functions *one* and *two*, respectively, are used as argument. Furthermore, we extend the alphabet of the resulting predicate with tr and the state components; these are all the names that are in the alphabet of $a1$ which are neither a UTP observational variable nor *dashed*. Finally, we sequentially compose the parallel execution of both actions with the merge function $MPar$.

Although rather long, the mechanisation of the parallel composition has a direct correspondence to its semantics presented in Section 3.1. The same direct correspondence happens to the mechanisation of interleaving, hiding, parametrised actions and substitutions, which are omitted here, but can be found in [71]. Furthermore, the mechanisation of recursion is trivially defined in terms of the weakest fixed-point described in Section 3.2.3.

The *Circus* assignment is reactive, and hence, it needs a definition different from that of relational assignment. The *Circus* assignment also receives a *homogeneous* alphabet a that contains at least all the UTP observational variables and their *dashed* counterparts, a sequence ns of names and a sequence $exps$ of expressions. The same restrictions from the relational assignment apply: all the names in ns and free-variables in $exps$ must be *undashed* and belong to a; both lists ns and $exps$ have the same length. The set of tuples $(a, ns, exps)$ that satisfy these conditions is WF_Assign_C.

The reactive design that is returned in the definition below has $True_R\ a$ as its precondition; its postcondition states that the trace is left unchanged and that the final value of *wait* is *false*. Furthermore, we use the relational assignment to express the change of the state components accordingly.

$$
\begin{array}{|l}
Assign_C : WF\_Assign_C \rightarrow CSP\_PROCESS \\
\hline
\forall\, a : ALPHABET;\ ns : \mathrm{seq}\, VAR\_NAME;\ exps : \mathrm{seq}\, EXPRESSION \\
\quad \mid (a, ns, exps) \in WF\_Assign_C \\
\quad \bullet\ Assign_C(a, ns, exps) = \\
\qquad\quad R(\ (True_R\ a) \\
\qquad\qquad \vdash_D ((=_R (a, dash\, tr, Var(tr))) \wedge_R (=_R (a, dash\, wait, Val(Bool(false)))) \\
\qquad\qquad\quad \wedge_R Assign_R(a, ns, exps))\ )
\end{array}
$$

The specification statement $f : [preC, postC]$ receives a *homogeneous* alphabet a that contains, among other variables, all the UTP observational variables and their *dashed* counterparts, a sequence f of names, the precondition $preC$, and the postcondition $postC$. The set $WF\_SpecStatement_C$ is the set of all $(a, f, preC, postC)$ such that: every name in f is *undashed*, different from all of the UTP observational variables, and belongs to a; and the alphabets of $preC$ and $postC$ are equal to a.

$$
\begin{array}{|l}
SpecStatement_C : WF\_SpecStatement_C \rightarrow CSP\_PROCESS \\
\hline
\forall\, a : ALPHABET;\ f : \mathrm{seq}\, VAR\_NAME;\ preC : CIRCUS\_CONDITION; \\
\quad postC : CIRCUS\_PREDICATE \\
\qquad \mid (a, f, preC, postC) \in WF\_SpecStatement_C \\
\qquad \bullet\ SpecStatement_C(a, f, preC, postC) = \\
\qquad\qquad R(\ preC \\
\qquad\qquad\quad \vdash_D \\
\qquad\qquad\quad ((=_R (a, dash\, tr, Var(tr))) \wedge_R (=_R (a, dash\, wait, Val(Bool(false)))) \\
\qquad\qquad\quad\ \wedge_R postC \\
\qquad\qquad\quad\ \wedge_R \Pi_R(a \setminus ((\mathrm{ran}\, f \cup \{n : \mathrm{ran}\, f \bullet dash\, n\}) \cup ALPHABET\_CSP)))\ )
\end{array}
$$

As expected, the reactive design which is returned has $preC$ as its precondition; as for assignment, on termination, the specification statement does not change the trace and terminates. Furthermore, it also establishes the postcondition $postC$. Finally, the specification statement cannot change any variable that is not in the frame f; we mechanise this property using the relational skip on the alphabet that does not contain any observational variable and any variable that is in the frame (and their *dashed* counterpart).

The next *Circus* action whose mechanisation we present is the schema expression. As in Section 3.1, we also assume that schema expressions have already been normalised. However, a very important aspect is implicitly considered in Section 3.1 and must be made explicit in the mechanisation: the typing of the declared variables. The recursive function *Typing* receives a list of variable declarations and an alphabet, and returns a conjunction of predicates: for each variable n declared to be of type T, it contains a predicate $x \in_R T$. For instance, given the declaration $x : \mathbb{Z};\ y : \mathbb{Z}$, the result of $Typing\,((\langle x, y\rangle, \langle Val(Set(\mathbb{Z})), Val(Set(\mathbb{Z}))\rangle), \{x, y\})$ is the following mechanisation for the predicate $x \in \mathbb{Z} \wedge y \in \mathbb{Z}$.

$$
\begin{array}{l}
(=_R (\{x, y\}, Rel((\_ \in_R \_), Var(x), Val(Set(\mathbb{Z}))), Val(Bool(true)))) \\
\wedge_R (=_R (\{x, y\}, Rel((\_ \in_R \_), Var(y), Val(Set(\mathbb{Z}))), Val(Bool(true))))
\end{array}
$$

Notice that the first argument of the function *Typing*, the variable declaration, is a pair of lists: the first element is the list of variable names, which must be elements of the alphabet a, and the second element is the list of types; both lists must have the same size. These restrictions are captured by the type VAR_DECLS, whose cartesian product with $ALPHABET$ is the domain of the function $Typing$.

$$Typing : (VAR\_DECLS \times ALPHABET) \pfun REL\_PREDICATE$$

The set of well-formed schema expressions, $WF\_SchemaExp_C$, is the set containing all pairs $(decls, p)$, where $decls$ are the well-formed variable declarations, and p is a relational predicate, such that the set of all variables that are declared in $decls$ is equal to the alphabet of p removing the observational variables in $ALPHABET\_CSP$. For us, a schema expression is defined like a specification statement: the alphabet contains all the declared variables and the observational variables, the frame contains the *undashed* versions of all the *dashed* declared variables, the precondition is the existential quantification of the *dashed* variables, where the predicate also includes the typing restrictions of the variables, and the postcondition is the conjunction of the typing restrictions of the variables and p.

$$
\begin{array}{|l}
SchemaExp_C : WF\_SchemaExp_C \rightarrow CSP\_PROCESS \\
\hline
\forall\, decls : VAR\_DECLS;\ p : REL\_PREDICATE \mid (decls, p) \in WF\_SchemaExp_C \\
\quad \bullet \exists\, f : \operatorname{seq} VAR\_NAME \\
\qquad \mid \operatorname{ran} f \subseteq undashed \land \operatorname{ran}(decls.1 \upharpoonright dashed) = dash(\!|\ \operatorname{ran} f\ |\!) \\
\qquad \bullet\ SchemaExp_C(decls, p) = \\
\qquad\quad SpecStatement(\ \operatorname{ran} decls.1 \cup ALPHABET\_CSP, f, \\
\qquad\qquad\qquad \exists_R(\operatorname{ran}(decls.1) \setminus undashed, Typing(decls, p.1) \land_R p), \\
\qquad\qquad\qquad Typing(decls, p.1) \land_R p\ )
\end{array}
$$

Three auxiliary recursive functions are used in the mechanisation of our last command presented in this section, alternation. The three of them receive an element of $GUARDED\_ACTIONS$ as argument. This type contains all the pairs of finite lists with same length, in which the first element is a list of *Circus* conditions (the guards) and the second element is a list of actions. For example, the guarded actions $g_1 \rightarrow A_1 [\!] \ g_2 \rightarrow A_2$ is represented in our mechanisation as the pair $(\langle g_1, g_2\rangle, \langle A_1, A_2\rangle)$. The first function, $ValidGuards$, mechanises the predicate $\bigvee i \bullet g_i$; it returns the disjunction of all guards in the first list. The function $NonDivActions$ mechanises the predicate $\bigwedge i \bullet g_i \Rightarrow \neg\, A_{i\,f}^{\,f}$. Finally, the function $ExecActions$ mechanises the predicate $\bigvee i \bullet g_i \land A_{i\,f}^{\,t}$.

An alternation does not diverge if at least one of the guards is valid ($ValidGuards$) and if every action guarded by a valid guard does not diverges ($NonDivActions$). When it terminates, it establishes the result of executing one of the actions that are being guarded by a valid guard ($ExecActions$).

$$
\begin{array}{|l}
if_C\_fi_C : GUARDED\_ACTIONS \rightarrow CSP\_PROCESS \\
\hline
\forall\, gactions : GUARDED\_ACTIONS \bullet \\
\qquad if_C\ gactions\ fi_C = R(\ ValidGuards(gactions) \land_R NonDivActions(gactions) \\
\qquad\qquad\qquad\qquad \vdash_D ExecActions(gactions)\ )
\end{array}
$$

In order to simplify proofs, we also provide a simpler binary alternation.

The semantics of all *Circus* processes are given as syntactic transformations from the process definition to some *Circus* action. Their mechanisation is left as future work.

3.3 Final Considerations

This chapter presented *Circus*'s denotational semantics and its mechanisation in a theorem prover, ProofPower-Z. In the denotational semantics, we took the approach from [31], where the semantics of the CSP operators are given as reactive designs. By expressing the vast majority of the *Circus* operators as reactive designs, we reuse the results presented in [31], bring uniformity to proofs, and foster reuse of our results. Furthermore, we believe that the definitions of the operators as reactive designs provided us with simpler and more intuitive definitions.

Although based on the work presented in [105], the denotational semantics we presented in this chapter has some major differences. For instance, the semantics presented in [105] did not allow us to prove our refinement laws because it was a shallow embedding of *Circus* in Z. This was our main motivation for defining a new denotational semantics for *Circus*.

The semantic model for *Circus* processes presented in [105] was a Z specification. For this reason, the state invariant was implicitly maintained by all operators. In our semantics, this is no longer a fact: nothing is explicitly stated about the invariant in our semantics. We assume specifications that initially contain no command, and therefore, change the state using only Z operations, which explicitly include the state invariant and guarantee that it is maintained. For this reason, our semantics ignores any existing state invariants, since they are considered in the refinement process, just as in Z.

As a direct consequence of our definition for external choice and the need for $Stop$ to be its unit, our semantics of $Stop$ does not keep the state unchanged, but loose. An alternative would be to allow state changes to resolve the choice, in which case, $Stop$ would keep the state unchanged. However, the states of the processes are encapsulated and state changes should not be noticed by the external environment; for this reason, we chose the first approach.

Another major difference from the semantics presented in [105] is the state partitions in parallel composition and interleaving, which remove the problems intrinsic to shared variables. These partitions were originally introduced in [27], and also have a direct consequence in the semantics of parallel composition and interleaving of processes. In [105], the parallel composition $P \llbracket cs \rrbracket Q$ conjoins each paragraph in P (Q) with $\Delta Q.State$ ($\Delta P.State$); this lifts the paragraphs in P (Q) to a state containing also the elements of Q (P), but with no extra restrictions. For us, in the semantics of parallel composition and interleaving, each side of the composition has a copy of all the variables in scope. They may change the values of all these variables, but only the changes to those variables that are in their partition have an effect in the final state of the composition. For this reason, we do not need to leave $Q.State$ unconstrained. We use a definition that is very similar to the other binary process combinators; the only change is the consideration of state partitions.

For most of the *Circus* operators, the fact that they are **R** and **CSP** healthy follows directly from their definitions and from the fact that reactive designs are **CSP1** and **CSP2** [31]. Those which were not defined as reactive designs were also proved to be **R** and **CSP** healthy. However, process that can be defined using the CSP notation also satisfy healthiness conditions, which are expressed in terms of CSP constructs; in this chapter, we lifted these conditions to *Circus*, giving rise to the healthiness conditions **C1** and **C2**. A final healthiness condition was also needed for *Circus* actions, **C3**. It states that, when expressed as a reactive design, every *Circus* action does not contain any *dashed* variable in its precondition.

We started our way towards a theorem prover for *Circus* by giving a set-based model to relations. This is the basis for the development of five theories: relations, designs, reactive processes, CSP processes, and *Circus* processes. For us, a relation is a pair, whose first element is a set that represents its alphabet and whose second element is a set of functions from names to values; the domain of all these functions are equal to the relation alphabet.

This is not the only possible model for relations. Our choice was based on the fact that any restriction that applies to relations has a direct impact on the complexity of the proofs. Our model imposes a simple restriction: the domain of the bindings must be equal to the alphabet. This restriction results in simpler definitions, and hence proofs. As an example of an alternative, in [32] a relation is defined as a pair formed by an alphabet and a set of pairs of bindings: for every pair (b_1, b_2) of bindings in a relation, the domain of b_1 has only *undashed* names and that of b_2 only *dashed* names. Such a restriction has to be enforced by the definition of every operator. There is, however, an isomorphism between our model and this one. By joining and splitting the sets of bindings, we can move from one model to another; our concern is only with the practicality of mechanical theorem proving.

We also could have used bindings whose domains could be different from the relation's alphabet. However, the alphabet is the set of names constrained by the relation. Hence, the alphabet a of a relation would have to be either a subset or equal to the domain of each binding b. Values of names that were not in the alphabet would actually have no meaning. We chose bindings whose domain is the alphabet because, by taking the other approach, we have a more complex definition for alphabet extension: bindings for names that are not in the alphabet need to be removed before being left unrestricted. Alphabet extension is at the heart of the definitions of conjunction and disjunction.

If, in the hope to find simplifications in other points, we accepted the more complex definition of alphabet extension, then we would need to determine how to handle the names that are not in the alphabet of the relation. For example, bindings could be total functions that map these names to an undefined value $\perp$; or we could leave these names unrestricted. These restrictions on relations are in fact more complex than that in our model, and lead to more complex definitions and proofs. We also have an isomorphism between our model and each of these; by applying a domain restriction to the bindings in these models and extending our model's bindings, we can change the representations.

As an industrial theorem prover, ProofPower-Z proved to be powerful (and helpful). The support provided by hundreds of built-in tactics and theories, as libraries for Z constructs and set theory, made our work much simpler. The axiomatisation of the theorems proved in our work in other theorem provers, like Z/Eves, and the development of new theories based on these axioms makes the use of our results in different theorem provers possible. In ProofPower-Z, the tactics that can be created are more powerful than the tactics available in Z/Eves; however, the level of expertise needed for initial users of Z/Eves is not as high as for ProofPower-Z.

The discussion above of alternative models is based on our experience with ProofPower-Z; some of them could make proofs easier in another theorem prover. An investigation of alternative theorem provers is a topic for future research.

Nuka and Woodcock formalised the alphabetised relational calculus in Z/EVES [67]. They did not restrict the set of bindings in the same way we do, but the restriction on the domain of the bindings is satisfied by all the constructors. By including the restriction on the set of bindings, we make this information available in all the proofs, and not only in those including some particular operators. We extend [67] by including many other operations, such as sequencing, assignment, refinement, and recursion. The hierarchical mechanisation of the theories of designs, reactive processes, CSP, and *Circus* is also a contribution of our work that provides a powerful tool for further investigations on them.

In [68], the authors present the same mechanisation that was presented in [67] but, this time, in ProofPower-Z. They also extend [67] by mechanising a specification language that includes, among other operators, skip, abort, miracle, Hoare triples, assertions, coercions, weakest preconditions, and iterations. However, their syntax is defined using Z free types; as discussed in this chapter, this makes it harder to extend their specification language.

Hoare and He [54], although dealing with alphabetised predicates, often leave alphabets quite implicit. For example, *true* is often seen unalphabetised, while in fact, it is alphabetised. This abstraction simplifies the presentation of the theory, but is not suitable for theorem provers. With the obligation to deal with alphabets, our work gives more details on how the alphabets are handled within the UTP.

The alphabet extension used in the UTP constrains the values of the new variables: they cannot be changed. However, our set-based model for relations needed a different alphabet extension that leaves their values unconstrained. Furthermore, in the UTP, existential quantifications are used in two different ways: in the definition of variable blocks, the authors explicitly state that the quantified variables are removed from the alphabet; and in the definition of the reactive *SKIP*, the alphabet is, implicitly, left unchanged. Our implementation defines two existential and two universal quantifications: one of them removes the quantified variables from the alphabet, and the other one does not. We also changed the formulation of some of the UTP definitions in order to facilitate our proofs; the relational sequence is an example of such definition.

Our work also reveals details that are left implicit in the UTP regarding the domain of the healthiness conditions. By mechanising the healthiness conditions, **R3** for instance, we make it explicit that **R3**, and consequently **R**, is a partial function that can only be applied to *homogeneous* reactive processes.

We expressed the language constructors as functions. For this reason, they can be simply extended without losing the previous proofs; the syntax of expressions was abstracted by using three simple definitions. Furthermore, the strategy that we adopted for lifting Z functions and relations to relational predicates, for instance $\leq_R$, makes the Z toolkit directly available in our theory.

The mechanisation of the CSP and *Circus* theory proved to be harder than we first imagined. Some simple expressions proved to be non-trivial when it came to the mechanisation. For instance, in the mechanisation of the function do_C, the expressions regarding the refusal set and the increment of the trace, and the representation of events were not trivial. However, our strategy for lifting Z functions and relations to values proved to be of much use in both cases.

Another interesting topic was raised in the mechanisation of schema expressions. Implicitly, in the denotational semantics, the type of the variables is already considered; however, in the mechanisation, we have to make this explicit. In our mechanisation, we defined a function that characterises a relational predicate that imposes the typing of the variables.

In [45, 106], Freitas *et al.* present a model checker for *Circus* that will be integrated to our theorem prover. Furthermore, an operational semantics for *Circus* is also presented; proving the correspondence between our semantics presented in this chapter and that in [45] using the method presented in [54] is an interesting piece of future work.

Our aim is to provide a mechanisation of the UTP that can support the development of other languages theoretically based on the UTP. *Circus* is such a language, and is the first one to use our mechanisation of the UTP. In this chapter, we presented a mechanisation of the *Circus* theory, which is based on the CSP theory, and mechanises the final version of the semantics of *Circus*. In the next chapter, we present the refinement notions and laws of *Circus*. This includes the proofs of the refinement laws proposed so far, whose manual proofs we intend to automate in the near future providing *Circus* with a mechanised refinement calculus that can be used in the formal development of state-rich reactive programs.

Chapter 4

Refinement: Notions and Laws

In this chapter we discuss the refinement notions for *Circus* processes and their constituent actions. The simulation technique, a refinement strategy for the development of centralised specifications into distributed implementations, and some laws presented in [27] are also discussed. Furthermore, new refinement laws are presented in this chapter. Finally, we present some of the proofs that show that the *Circus* refinement laws proposed by us are sound with respect to its semantics presented in Chapter 3.

4.1 Refinement Notions and Strategy

The central notion in the UTP is refinement, which is expressed as an implication: an implementation P satisfies a specification S if, and only if, $[P \Rightarrow S]$. The square brackets denote the universal quantifier over the alphabet, as in [38], which must be the same for implementation and specification. In *Circus*, the basic notion of refinement is that of action refinement [26, 84].

Definition 4.1 (Action Refinement) *For actions A_1 and A_2 on the same state space, the refinement $A_1 \sqsubseteq_{\mathcal{A}} A_2$ holds if, and only if, $[A_2 \Rightarrow A_1]$.* ❑

The action refinement relation is a partial order and the action constructors are monotonic with respect to it. Hence, we can adopt a piecewise and stepwise refinement technique.

For processes, since we have that the state of a process is private, we have a slightly different definition. Basically, the main action of a process defines its behaviour. For this reason, process refinement is defined in terms of action refinement of local blocks. In the following, $P_1.State$ and $P_1.Act$ denote the local state and the main action of process P_1; similarly for process P_2.

Definition 4.2 (Process Refinement) *$P_1 \sqsubseteq_{\mathcal{P}} P_2$ if, and only if,*
$(\exists P_1.State;\ P_1.State' \bullet P_1.Act) \sqsubseteq_{\mathcal{A}} (\exists P_2.State;\ P_2.State' \bullet P_2.Act)$ ❑

The actions $P_1.Act$ and $P_2.Act$ may act on different state spaces and their dashed counterparts, and so may not be comparable. Actually, we compare the actions we obtain by hiding the state components of processes P_1 and P_2, as if they were declared in a local

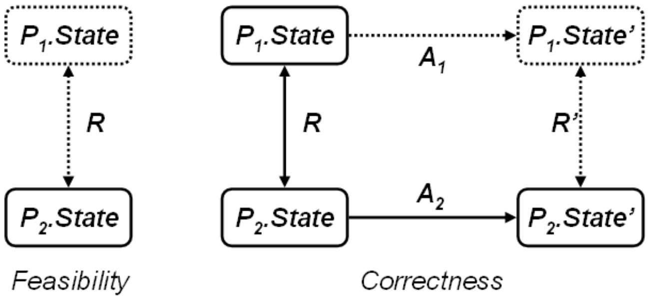


Figure 4.1: Forwards Simulation

variable block, whose semantics is given by existential quantification. We are left with a state space containing only the UTP observational variables *okay*, *wait*, *tr*, and *ref*.

As discussed above, the state of a process is private. This allows processes' components to be changed during a refinement. This can be achieved in much the same way as we can data refine variable blocks and modules in imperative programs [64]. A well-known technique of data refinement in those contexts is forwards simulation [56].

In [27], the standard simulation techniques used in Z were adopted to handle processes and actions. A simulation is a relation between the states of two processes that satisfies a number of properties.

Definition 4.3 (Forwards Simulation) *A forwards simulation between actions A_1 and A_2 of processes P_1 and P_2, with local state L, is a relation R between $P_1.State$, $P_2.State$, and L satisfying*

1. *(Feasibility)* $\forall P_2.State;\ L \bullet (\exists P_1.State \bullet R)$
2. *(Correctness)* $\forall P_1.State;\ P_2.State;\ P_2.State';\ L \bullet$
$$R \land A_2 \Rightarrow (\exists P_1.State';\ L' \bullet A_1 \land R') \qquad \square$$

We write $A_1 \preceq_{P_1,P_2,R,L} A_2$ to denote such a simulation; we omit the subscripts when they are clear from the context. A forwards simulation between P_1 and P_2 is a forwards simulation between their main actions.

In Figure 4.1, we illustrate both properties. The *feasibility* property guarantees that for every initial concrete state $P_2.State$ there exists an initial abstract state $P_1.State$ that can be reached via the retrieve relation R. The *correctness* property guarantees that for every abstract and concrete states $P_1.State$ and $P_2.State$, connected by R, and for every concrete state $P_2.State'$ resulting from the execution of A_2 in the state $P_2.State$, there must exist a final abstract state $P_1.State'$, which is the result of the execution of A_1 in the state $P_1.State$, and is connected by R' to $P_2.State'$.

Notice that, in Definition 4.3, no applicability requirement concerning preconditions exists. This follows from the fact that actions are total. If an action is executed outside its precondition, it diverges; arbitrary new synchronisation and communication can be observed, but no past observation is affected. Furthermore, no specific condition is imposed on the initialisation: state initialisations are explicitly included in the main action.

A theorem presented in [27] and proved in [71], ensures that, if we provide a forwards simulation between two processes P_1 and P_2, we can substitute P_1 for P_2 in a program.

Theorem 4.1 (Forwards simulation is sound) *When a forwards simulation exists between two processes P_1 and P_2, we also have that $P_1 \sqsubseteq_{\mathcal{P}} P_2$.* ❑

A refinement strategy for *Circus* has already been presented [27]. This strategy, although simple, can effectively serve as a tool to guide and transform an abstract (usually centralised) specification into a concrete (usually distributed) solution of the system implementation. This strategy is based on laws of simulation, and action and process refinements, which are presented in Appendix C. We, however, present further simulation and refinement laws.

Each iteration within the refinement strategy, which may include many iterations, includes three steps: simulation, action refinement, and process refinement. Figure 4.2, taken from [27], summarises one of these iterations. The first two steps are used to reorganise the internal structure of the process: we use simulation to introduce the elements of the concrete system state, and then, the actions are refined in order to be partitioned in a way that each partition operates on different components of the modified state. These changes result in the splitting of the state space and the accompanying actions into two different partitions, in such a way that each partition groups some state components and the actions that access these components. After the second step, we have a structure in which each partition clearly has an independent state and behaviour. The third step of the strategy upgrades each of these partitions to individual processes: the resulting processes are combined in the same way as their main actions were in the previous process (see Chapters 1 and 5 for examples).

4.2 Laws of Simulation

In order to carry the data refinement in a stepwise way, some laws of simulation are provided. These laws provide support to prove that a relation R is indeed a forwards simulation. Besides, they can be used to prove simulations for schema actions, in much the same way as we do in Z.

Simulation distributes through the primitive actions *Skip*, *Stop*, and *Chaos*. The simulation of schemas raises the same provisos as in the standard Z rule. The law C.4 presented below includes an applicability condition, which does not appear in the definition of forwards simulation, since it is concerned with the semantics of actions, which are total operations on the state that include the UTP variables. A schema expression, on the other hand, is an operation over the process state, and it is not total.

Law C.4 (Schema expressions). $ASExp \preceq CSExp$
provided

- ➩ $\forall P_1.State;\ P_2.State;\ L \bullet R \land \mathrm{pre}\ ASExp \Rightarrow \mathrm{pre}\ CSExp$
- ➩ $\forall P_1.State;\ P_2.State;\ P_2.State';\ L \bullet$
  $R \land \mathrm{pre}\ ASExp \land CSExp \Rightarrow (\exists P_1.State';\ L' \bullet R' \land ASExp)$ ❑

Forwards simulation distributes through the other constructs. In the following, we

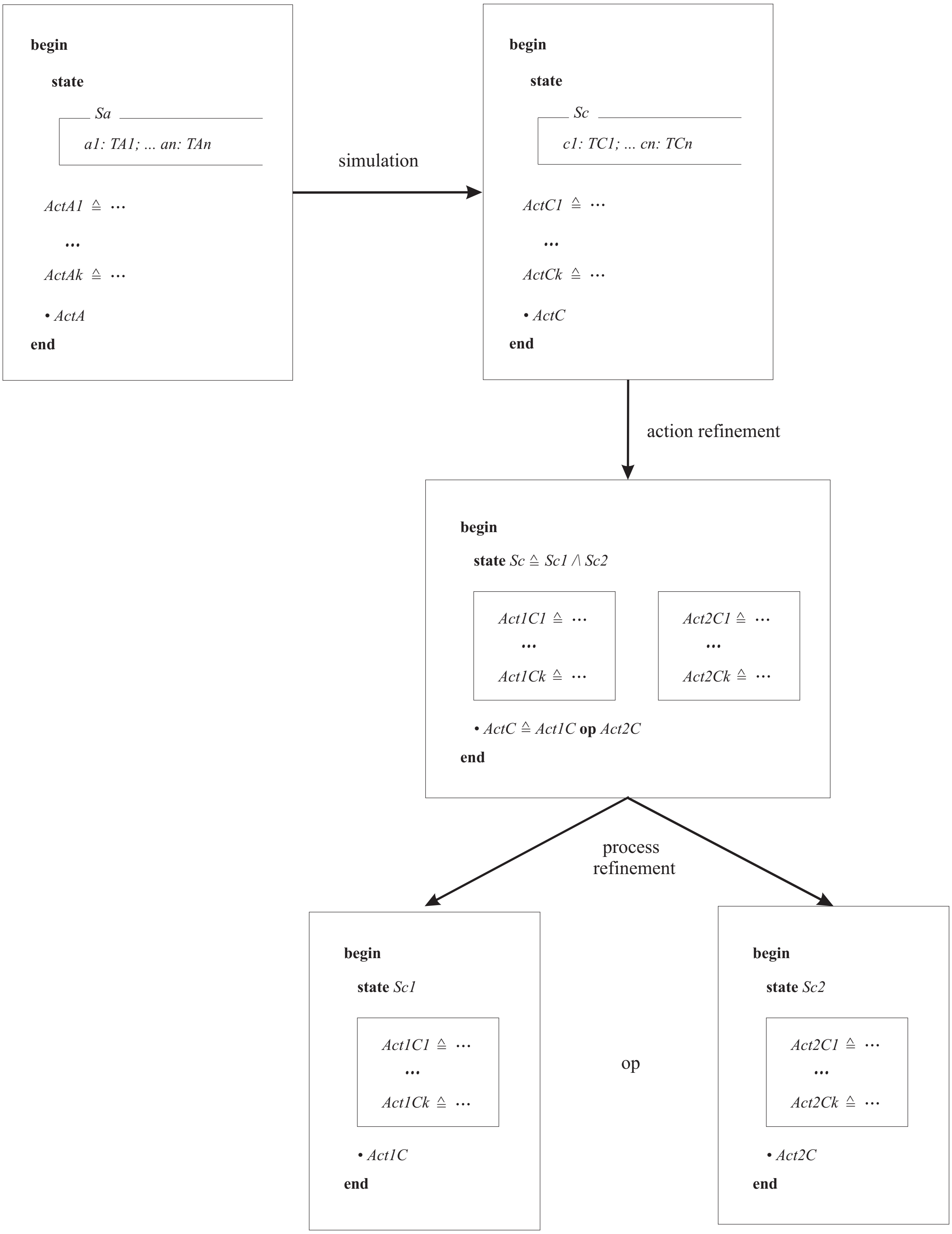


Figure 4.2: An iteration of the refinement strategy

present some of the distributions laws. The distribution of simulation through external choice was first proposed in [27]; however, an important restriction on the retrieve relation was not considered.

Law C.13 (External choice distribution*). $A_1 \Box A_2 \preceq B_1 \Box B_2$
provided

- ➭ $A_1 \preceq B_1$
- ➭ $A_2 \preceq B_2$
- ➭ *R is a function from the concrete to the abstract state* ❑

Besides having the expected requirements regarding the simulation of each of the branches, the distribution of simulation through an external choice between two arbitrary actions also requires the retrieve relation to be a function from the concrete to the abstract state, as in [57]. Intuitively, if different abstract values correspond to the same concrete value, such values could have been merged in the abstraction; "abstract enough" abstractions would not have non-functional retrieve relations (from concrete to abstract).

The following example illustrates the need for the restriction on the retrieve relation. Let us have an abstract process P_A with a non-empty sequence $OnOffSq : \mathrm{seq}\, 0..1$ as the only state component. Now, let us consider a data refinement to a concrete process P_C with a state component $OnOff : 0..1$, using the retrieve relation presented below. The function *head* returns the head of a given non-empty list.

$$Ret \mathrel{\widehat=} [OnOffSq : \mathrm{seq}\, 0..1;\ OnOff : 0..1 \mid OnOffSq \neq \langle\rangle \land OnOff = head(OnOffSq)]$$

Now, suppose we have the following action in P_A.

$$OnOffSq := \langle 0 \rangle;\ ((c.OnOffSq \to Skip) \Box (c.OnOffSq \frown \langle 1 \rangle \to Skip))$$

Each of the branches in the choice are individual simulations of the following action.

$$c.\langle OnOff \rangle \to Skip$$

We have no actual choice happening in P_C. Hence, $\langle (c, \langle 0 \rangle) \rangle$ and $\langle (c, \langle 0, 1 \rangle) \rangle$ are possible traces of the abstract action, but only $\langle (c, \langle 0 \rangle) \rangle$ is a possible trace of the concrete action; although the individual concrete branches are refinements of the corresponding abstract ones, we do not have an overall refinement.

The restriction on the retrieve relation may be relaxed if, for instance, we guarantee that the choices are on different channels, as we present in the following law.

Law C.14 (External choice/Prefix distribution*).

$$\Box\, i \bullet c_i \to A_i \preceq \Box\, i \bullet c_i \to B_i$$

provided $\forall\, i \bullet A_i \preceq B_i$ ❑

Parallel actions work on disjoint parts of the state: no interference occurs. This fact is used in the simulation law for the parallel operator. In [27], the restrictions on the state partitions of the parallel operator were not considered; hence, we present a new version of this law below.

Law C.21 (Parallel composition distribution*).

$$A_1 \llbracket ns_{1_A} \mid cs \mid ns_{2_A} \rrbracket A_2 \preceq B_1 \llbracket ns_{1_B} \mid cs \mid ns_{2_B} \rrbracket B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$
- $\forall v_A, v_B \bullet R(v_A, v_B) \Rightarrow (v_A \in ns_{1_A} \Rightarrow v_B \in ns_{1_B})$
- $\forall v_A, v_B \bullet R(v_A, v_B) \Rightarrow (v_A \in ns_{2_A} \Rightarrow v_B \in ns_{2_B})$ ❑

The last two proof obligations guarantee that if a component is in a partition in a abstract parallel composition then it is in the corresponding partition in a concrete parallel composition.

Further new simulation laws are straightforward; for conciseness, they are omitted here, but can be found in Appendix C. They complete the evidence for the claim that simulation distributes through the structure of actions by considering the prefix operator of a synchronisation event c, internal choice, and interleaving.

In [27], the law for simulation of recursive actions is not strong enough to support distribution. It considers part of the recursive body, but not all of it. Nevertheless, simulation does distribute through recursion if the concrete function is simulated by the abstract one, as we state in the following law.

Law C.23 (Recursion distribution*). $\mu X \bullet F(X) \preceq \mu X \bullet F'(X)$
provided $F \preceq F'$ ❑

The above law is what is actually used in our case study.

4.3 Action Refinement

In the second step of the refinement strategy, an algorithmic refinement on actions is proposed. This action refinement is justified by the following theorem, which is proved in [71].

Theorem 4.2 (Soundness of action refinement) *Suppose we have a process P with actions A_1 and A_2. If $A_1 \sqsubseteq_{\mathcal{A}} A_2$, then the identity is a forwards simulation between A_1 and A_2.* ❑

In this definition, and in the following, we make no distinction between the *Circus* action and its UTP semantics.

In the following, we present corrections for some of the laws presented in [27] and some of the new laws required by our case study presented in Chapter 5. They are samples of some groups of laws: assumptions, guards, schemas, parallel composition, prefix, external choice, hiding, and commands.

4.3.1 Laws of Assumptions

Most of the laws of assumptions are novel. They are very useful in the refinement process, because they allow us to record information about the state of the process that is needed in many points.

Our first law allows us to compose any assumption in sequence with a weaker one.

Law C.27 (Assumption Introduction*). $\{g\} = \{g\};\ \{g_1\}$
provided $g \Rightarrow g_1$ ❑

Our next law states that we may assume that g is valid after a guard g.

Law C.31 (Guard/Assumption—introduction 1*). $g \,\&\, A = g \,\&\, \{g\};\ A$

Our case study [75] was defined using mutually recursive actions. During its development, we needed to distribute assumptions over mutually recursive actions, which can be expressed as follows.

$$\mu\, X_1, \ldots, X_i, \ldots, X_n \bullet \left\langle \begin{array}{l} F_1(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_i(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_n(X_1, \ldots, X_i, \ldots, X_n) \end{array} \right\rangle$$

The Law C.40 that follows states that if an assumption g is valid before a mutual recursion, it is also valid before the i-th action F_i, provided no action invalidates g before any recursive invocation. In the law presented below, the angled brackets denote a vector of fixed points. Furthermore, the law does not impose the start of the execution in any action; we omit the index of the initial action.

Law C.40 (Assumption/Mutual recursion—distribution*).

$$\{g\};\ \mu\, X_1, \ldots, X_i, \ldots, X_n \bullet \left\langle \begin{array}{l} F_1(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_i(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_n(X_1, \ldots, X_i, \ldots, X_n) \end{array} \right\rangle$$

$$\sqsubseteq_{\mathcal{A}}$$

$$\mu\, X_1, \ldots, X_i, \ldots, X_n \bullet \left\langle \begin{array}{l} F_1(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ \{g\};\ F_i(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_n(X_1, \ldots, X_i, \ldots, X_n) \end{array} \right\rangle$$

provided for all j, such that $1 \leq j \leq n$,

$$\{g\};\ F_j(X_1, \ldots, X_i, \ldots, X_n) \sqsubseteq_{\mathcal{A}} F_j(\{g\};\ X_1, \ldots, \{g\};\ X_i, \ldots, \{g\};\ X_n),$$

❑

The following example illustrates an application of this law. It moves an assumption that a variable y has the value 0 to the first action of the recursion. The fact that y is changed to 1 if an event c_2 happens does not invalidate the refinement because, after c_3 happens the value of y is set to zero again before the recursive invocation.

$$\begin{array}{l}\{y = 0\};\ \mu X_1, X_2 \bullet \langle (c_1 \to X_1) \Box (c_2 \to y := 1;\ c_3 \to y := 0;\ X_2), c_3 \to X_1 \rangle \\ \sqsubseteq_{\mathcal{A}} \\ \mu X_1, X_2 \bullet \langle \{y = 0\};\ ((c_1 \to X_1) \Box (c_2 \to y := 1;\ c_3 \to y := 0;\ X_2)), c_3 \to X_1 \rangle\end{array}$$

provided

➪ $\{y = 0\};\ ((c_1 \to X_1) \Box (c_2 \to y := 1;\ c_3 \to y := 0;\ X_2))$
$\sqsubseteq_{\mathcal{A}}$
$(c_1 \to \{y = 0\};\ X_1) \Box (c_2 \to y := 1;\ c_3 \to y := 0;\ \{y = 0\};\ X_2)$

➪ $\{y = 0\};\ (c_3 \to X_1) \sqsubseteq_{\mathcal{A}} c_3 \to \{y = 0\};\ X_1$ ❑

Both provisos are trivial; however, they need refinement laws for distributing the assumption. Some of them are already presented in [27]; others are introduced here. For instance, the next laws allow the distribution of an assumption through the prefix operator. First, we have that it distributes through output prefix.

Law C.45 (Assumption/Output prefix—distribution*).

$$\{g\};\ c!x \to A \sqsubseteq_{\mathcal{A}} c!x \to \{g\};\ A$$

This law is valid because *Circus* specifications do not mention any of the UTP variables. Otherwise, g could mention some property on the traces that could not be valid after the event c; this would invalidate the refinement.

Next, we have that an assumption also distributes through an input prefix. However, in order to avoid capture, the input variable x cannot occur in g.

Law C.47 (Assumption/Input prefix—distribution*).

$$\{g\};\ c?x \to A \sqsubseteq_{\mathcal{A}} c?x \to \{g\};\ A$$

provided $x \notin FV(g)$ ❑

Finally, assumption also distributes through synchronisation events.

Law C.41 (Assumption/Prefix—distribution*). $\{g\};\ c \to A \sqsubseteq_{\mathcal{A}} c \to \{g\};\ A$

Our final law on assumptions allows assumptions to be moved over schemas.

Law C.53 (Assumption/Schema—distribution*).

$$\{g\};\ [decl \mid p] \sqsubseteq_{\mathcal{A}} [decl \mid p];\ \{g\}$$

provided $g \wedge p \Rightarrow g'$ ❑

Nevertheless, the schema predicate cannot invalidate the assumption.

4.3.2 Laws of Guards

First, we present a simple law that transforms an action guarded by a disjunction into an external choice of two guarded actions.

Law C.58 (Guards Expansion*). $(g_1 \lor g_2) \mathbin{\&} A = g_1 \mathbin{\&} A \mathbin{\Box} g_2 \mathbin{\&} A$

The action $(g_1 \lor g_2) \mathbin{\&} A$ behaves like A if any of g_1 or g_2 is valid. In the equivalent action, a choice is given between two actions guarded by g_1 and g_2; if any of these guards is valid, A is the available choice.

If a parallel composition is guarded by a conjunction of two guards g_1 and g_2, we may move each of these guards to a different side of the parallel composition, provided the two conjuncts are equivalent.

Law C.64 (Guards/Parallel composition—distribution 3*).

$$(g_1 \land g_2) \mathbin{\&} (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (g_1 \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \mathbin{\&} A_2)$$

provided $g_1 \Leftrightarrow g_2$ ❑

If both g_1 and g_2 are valid, both actions behave like the parallel composition of A_1 and A_2; otherwise, they deadlock.

4.3.3 Laws of Schemas

The laws of schemas are particularly interesting because they lie on the very intersection between the CSP and the Z parts of a *Circus* specification.

In the laws that follow, we use two more functions to specify the provisos of the laws: the function $usedV$ gives the set of used variables (read, but not written); the function $wrtV$ gives the set of variables that are written by a given action. For schema expressions, $wrtV$ gives the set of variables that are constrained by the schema. In this definition from [27], we use v' to denote the list of dashed variables $v'_1, \ldots, v'_n$ of dashed free-variables (DFV) of $SExp$. The undashed representation v denotes the corresponding list of undashed free-variables. Furthermore, the notation $v' : T$ stands for the sequence of declarations $v'_1 : T_1; \ldots; v'_n : T_n$, which declares each of the variables in v' with its corresponding type as defined in $SExp$.

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

We hide all the dashed free-variables of the $SExp$ and then, we declare them again; however, we restrict their values to be the same as the values of their corresponding undashed variables. If we get a different schema expression, that means that their values were actually changed in the original expression, and hence, they should belong to the set of written variables.

If we have a schema expression followed by a parallel composition, the following law specifies the conditions under which we may move it to one of the sides of the parallel composition.

Law C.73 (Schema/Parallel composition—distribution*).

$$SExp;\ (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (SExp;\ A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$$

provided

- ➯ $wrtV(SExp) \subseteq ns_1$
- ➯ $wrtV(SExp) \cap usedV(A_2) = \varnothing$ ❑

Moving the schema into the parallel composition is possible if the schema changes only variables declared in the corresponding partition of the parallel composition, and if the variables it writes are not used by the other side of the parallel composition. A similar law holds for interleaving.

The final law on schemas is concerned with refinement of schemas. The definition of this law and a few others are standard; they are omitted here for the sake of conciseness.

4.3.4 Laws of Parallel Composition

The parallel composition is commutative. Furthermore, new variables may be included in one of the partitions of a parallel composition if they do not belong to the other partition already.

Law C.77 (Partition expansion*).

$$\begin{aligned}&\mathbf{var}\ x : T \bullet A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)\\ &= \mathbf{var}\ x : T \bullet A_1;\ (A_2 \llbracket ns_1 \cup \{x\} \mid cs \mid ns_2 \rrbracket A_3)\end{aligned}$$

provided $x \notin ns_2$ ❑

Actually, the final value of x might be different in both sides of the refinement law: in the left-hand side, we have that the final value of x may be either that determined by A_1, if $x \notin ns_1$, or that determined by A_2, otherwise; in the right-hand side, the final value of x is that determined by A_2. However, since we have a variable block, this difference cannot be seen by any other action afterwards. If, we had an action A_4 within the variable block following the parallel composition, this law would not be valid; but this is not the case.

An event c prefixing an action A may be put in parallel with the same event prefixing $Skip$ as follows.

Law C.78 (Parallel composition introduction 1*).

$$c \to A = (c \to A \llbracket ns_1 \mid \{\!| c |\!\} \mid ns_2 \rrbracket c \to Skip)$$

provided

- ➯ $c \notin usedC(A)$
- ➯ $wrtV(A) \subseteq ns_1$ ❑

It is valid since the proviso ensures that A will not deadlock because it has no occurrences of c and that it does not update any variable in ns_2. This derives directly from the fact that, by definition, $ns_1 \cap ns_2 = \varnothing$, and hence, $wrtV(A) \subseteq ns_1 \Leftrightarrow wrtV(A) \cap ns_2 = \varnothing$. The function *usedC* returns a set of all channels mentioned in an action.

Communications may be introduced using the following law. It extends a parallel composition by communicating a value e from an action A_1 to an action $A_2(x)$ and replaces, in A_2, direct references to x by the expression e. The introduced communication must be already in the synchronisation set, and hidden from the environment. Furthermore, in order to avoid capture, the variable x cannot be already in use.

Law C.82 (Channel Extension 3*).

$$
\begin{array}{l}
(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2(e)) \setminus cs_2 \\
= \\
((c!e \rightarrow A_1) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (c?x \rightarrow A_2(x))) \setminus cs_2
\end{array}
$$

provided

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

In [27], the authors present a very useful law: the Parallel composition/Sequence—step law provides a way of moving one action that precedes a parallel composition to one of the sides of this composition. However, an important proviso regarding the state partition is not considered. The function *initials* gives a set containing all the events in which action A is initially willing to synchronise.

Law C.84 (Parallel composition/Sequence—step*).

$$
(A_1;\ A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3 = A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)
$$

provided

- $initials(A_3) \subseteq cs$
- $cs \cap usedC(A_1) = \varnothing$
- $wrtV(A_1) \cap usedV(A_3) = \varnothing$
- A_3 is divergence-free

These provisos guarantee that, because its initial events are in cs and it is divergence-free, A_3 cannot do anything apart from wait to synchronise. On the other hand, A_1 can work independently, since its channels are not in the synchronisation channel set. Finally, A_3 does not use any variable that is written by A_1. However, in the right-hand side of the law, the changes that A_1 does to the state components are not lost; they are the initial values of the parallel composition. However, in the left-hand side of the law, any change that A_1 does to any component that is in ns_2 will be lost since its side of the parallel composition does not have priority on the partition ns_2. Hence, this law still needs one more proof obligation; This last proviso guarantees that A_1 can indeed be moved to the left-hand side of the parallel composition because it writes only on variables that are in ns_1; therefore, any changes to state components will not be lost.

➩ $wrtV(A_1) \subseteq ns_1$ ❑

The next law can be used to introduce new choices to one side of a parallel composition.

Law C.86 (Parallel composition/External choice—expansion*).

$$(\Box\, i \bullet a_i \to A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\Box\, j \bullet b_j \to B_j)$$
$$=$$
$$(\Box\, i \bullet a_i \to A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket ((\Box\, j \bullet b_j \to B_j) \Box (c \to C))$$

provided

- $\bigcup_i \{a_i\} \subseteq cs$
- $c \in cs$
- $c \notin \bigcup_i \{a_i\}$
- $c \notin \bigcup_j \{b_j\}$

The provisos guarantee that no a_i can synchronise with c because they are different from c and all of them are in cs. Furthermore, c is different from any of the b_is; we are not introducing nondeterminism. This law is very useful when we want one of the branches to have a particular form.

An iterated external choice of guarded parallel actions, in which the right-hand side of the parallel composition is always the same action A, can be written as the parallel composition of the iterated external choice and the action A.

Law C.87 (Parallel composition/External choice—distribution*).

$$\Box\, i \bullet (A_i \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A) = (\Box\, i \bullet A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A$$

provided

➩ $initials(A) \subseteq cs$

➩ *A is deterministic.* ❑

In the external choice, A is executed in parallel with some A_i. In the equivalent parallel composition, the proviso guarantees that, initially, A is able to execute no independent event; for this reason, it is executed in parallel with some action A_i. The second proviso guarantees that no deadlock will occur in the left-hand side of the law, which would not occur in the right-hand side. By way of illustration, let us consider the following example;

for conciseness, we omit the state partitions, and the synchronisation channel set, which contains the channels c_1 and c_2.

$$
\begin{array}{l}
(c_1 \to \mathit{Skip}) \parallel ((c_1 \to \mathit{Skip}) \sqcap (c_2 \to \mathit{Skip})) \\
\Box\ (c_2 \to \mathit{Skip}) \parallel ((c_1 \to \mathit{Skip}) \sqcap (c_2 \to \mathit{Skip}))
\end{array}
$$

This action can indeed deadlock if the internal choice of the first branch chooses c_2 and the internal choice of the second branch chooses c_1. However, if we did not have the second proviso, we would be able to apply Law C.87 and obtain the following action.

$$((c_1 \to \mathit{Skip}) \Box (c_2 \to \mathit{Skip})) \parallel ((c_1 \to \mathit{Skip}) \sqcap (c_2 \to \mathit{Skip}))$$

Clearly, whatever branch is internally chosen in the right-hand side of the parallel composition, the synchronisation is still possible, and this action terminates successfully. A similar law (with stronger provisos) can be found in [27].

The next law states the conditions on which we may transform a sequence of parallel compositions into a parallel composition of sequences.

Law C.88 (Parallel composition/Sequence—distribution*).

$$
\begin{array}{l}
(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2);\ (B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2) \\
= \\
(A_1;\ B_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_2;\ B_2)
\end{array}
$$

provided

- $\mathit{initials}(B_1) \cup \mathit{initials}(B_2) \subseteq cs$
- $\mathit{usedC}(A_1) \cap \mathit{initials}(B_2) = \mathit{usedC}(A_2) \cap \mathit{initials}(B_1) = \varnothing$
- $\mathit{usedV}(B_1) \cap ns_2 = \mathit{usedV}(B_2) \cap ns_1 = \varnothing$ ❑

Basically, both B_1 and B_2 in the second parallel composition need to synchronise before proceeding. Furthermore, the initial events of B_1 and B_2 are not used by A_2 and A_1, respectively. These provisos guarantee that B_1 and B_2 will start only when both A_1 and A_2 are finished. A final proviso guarantees that B_1 is only concerned with the variables that belong to its partition ns_1; hence, whatever A_2 does to the other variables in ns_2 will not affect its behaviour. The same applies to B_2 and A_1.

The parallel composition of two actions that are willing to synchronise on some event in the synchronisation set, but have no common initial communications, deadlocks.

Law C.92 (Parallel composition Deadlocked 1*).

$$(c_1 \to A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \to A_2) = \mathit{Stop} = \mathit{Stop} \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \to A_2)$$

provided

- $c_1 \neq c_2$
- $\{c_1, c_2\} \subseteq cs$ ❑

The parallel composition is also deadlocked if one of the branches is already deadlocked (*Stop*) and the other one is waiting to synchronise on some event that will actually never happen.

4.3.5 Laws of Prefix

Any event c prefixing an action A may be replaced by the sequential composition of the event c prefixing $Skip$, and A.

Law C.100 (Prefix/*Skip).** $c \to A = (c \to Skip);\ A$

If an event c is not used within an action A, we may prefix A with c and hide this event from the environment.

Law C.103 (Prefix Introduction*). $A = (c \to A) \setminus \{\!|\ c\ |\!\}$
provided $c \notin usedC(A)$ ❑

The Prefix/External choice—distribution law was first proposed in [27].

Law C.104 (Prefix/External choice—distribution*).

$$c \to \Box\, i \bullet g_i \,\&\, A_i = \Box\, i \bullet g_i \,\&\, c \to A_i$$

provided

➪ $\vee\, i \bullet g_i$

The proviso guarantees that at least one of the guards is valid, and hence, the event c will indeed happen. However, if more than one guard is valid and the external choice $\Box\, i \bullet g_i \,\&\, A_i$ is deterministic, the distribution of the prefix through the external choice introduces a non-deterministic choice on c; one more proof obligation is needed. The proof obligation presented below states that two different guards cannot be valid at the same time; it guarantees that the nondeterminism will not be introduced.

➪ $\forall\, i, j \mid i \neq j \bullet \neg\, (g_i \wedge g_j)$ (guards are mutually exclusive) ❑

The next law on prefix allows us to distribute an input prefix over a parallel composition.

Law C.108 (Input prefix/Parallel composition—distribution*).

$$c?x \to (A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2) = (c?x \to A_1) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, (c?x \to A_2)$$

provided $c \in cs$ ❑

The proviso is that c must be in the synchronisation channel set cs.

4.3.6 Laws of External Choice

External choice is associative. Furthermore, we may remove a choice between two identical actions.

Law C.111 (External choice elimination*). $A \mathbin{\Box} A = A$

External choice of guarded actions may distribute through sequence.

Law C.112 (External choice/Sequence—distribution).

$$(\Box\, i \bullet g_i \mathbin{\&} c_i \rightarrow A_i);\ B = \Box\, i \bullet g_i \mathbin{\&} c_i \rightarrow A_i;\ B$$

Finally, *Stop* is the external choice zero.

4.3.7 Laws of Hiding

The following CSP laws of hiding are also valid in *Circus*, but were not presented in [27]. The first law states that hiding has no effect if the channel we are hiding is not being used by the action.

Law C.120 (Hiding Identity*). $A \setminus cs = A$
provided $cs \cap usedC(A) = \varnothing$ ❏

Yet another important law is the distribution of hiding through external choice.

Law C.122 (Hiding/External choice—distribution*).

$$(A_1 \mathbin{\Box} A_2) \setminus cs = (A_1 \setminus cs) \mathbin{\Box} (A_2 \setminus cs)$$

provided $(initials(A_1) \cup initials(A_2)) \cap cs = \varnothing$ ❏

The only proviso guarantees that we are not hiding the initial communication of any of the choices, and hence, introducing nondeterminism.

A new law of hiding expansion allows us to included channels in the hidden set of channels.

Law C.124 (Hiding expansion 2*). $A \setminus cs = A \setminus cs \cup \{c\}$
provided $c \notin usedC(A)$ ❏

These channels, however, may not be already in use by the action from which they are going to be hidden from.

Hiding also distributes through sequence; no proviso is needed.

Law C.125 (Hiding/Sequence—distribution*).

$$(A_1;\ A_2) \setminus cs = (A_1 \setminus cs);\ (A_2 \setminus cs)$$

However, in the distribution through parallel composition, we need to guarantee that we are not hiding any of the channels on which the parallel composition is synchronising.

Law C.127 (Hiding/Parallel composition—distribution*).

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = (A_1 \setminus cs_2) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (A_2 \setminus cs_2)$$

provided $cs_1 \cap cs_2 = \varnothing$ ❑

Finally, we have that *Chaos* is the zero for hiding.

4.3.8 Laws of Commands

Laws of commands are also very interesting because they are in the intersection of the CSP and the commands of a *Circus* specification. Before introducing the new refinement laws for commands in *Circus*, we present below a very important and useful theorem; it allows us to reuse the majority of the work done by Cavalcanti and Woodcock on the ZRC.

Theorem 4.3 *For every program P and Q in ZRC expressed using only* **skip**, *specification statements, assumptions, coercions, assignments, and sequences, if* $P \sqsubseteq Q$ *in ZRC, then* $P \sqsubseteq Q$ *also holds in the Circus refinement calculus.*

In what follows, we informally discuss the proof of this theorem, which is rather long and omitted here; it can be found in [71].

The semantics of ZRC is given using predicate transformers. Since we are not considering logical constants, these predicates transformers are universally conjunctive [29]. In [33], Cavalcanti and Woodcock present an isomorphism $pt2p$ between universally conjunctive predicate transformers and relations. First, we prove that $pt2p$ is monotonic with respect to refinement. Next, we consider $wp.z$ to be the semantics of any ZRC construct z expressed using only the constructs listed in Theorem 4.3. The relation r that corresponds to the semantics of z is given by the expression $pt2p(wp.z)$. Finally, the expression $\mathbf{R}(\mathbf{H1}(\mathbf{H2}(r)))$ is the reactive design that corresponds to the semantics of z. We conclude the proof by showing that the following equality holds for c being **skip**, specification statements, assumptions, coercions, assignments and sequences.

$$\mathcal{C}(c) = \mathbf{R}(\mathbf{H1}(\mathbf{H2}(pt2p(wp.c))))$$

Here, we use $\mathcal{C}$ to denote the *Circus* semantic function. ❑

As a direct consequence of this theorem, refinement laws for specification statements such as strengthening the post-condition and weakening the pre-condition, among others, may also be used in the *Circus* refinement calculus.

For the other commands, however, like alternations and variable blocks, we are not able to reuse the results presented in [29] because ZRC alternations and variable blocks are not defined in terms of *Circus* actions. Therefore, new refinement laws, some of which resemble the ones in ZRC, are needed.

In [27], the case study did not require any law for variable blocks. Our first new law regards the extension of variable blocks over parallel composition.

Law C.138 (Variable block/Parallel composition—extension*).

$$(\mathbf{var}\ x : T \bullet A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$$
$$=$$
$$(\mathbf{var}\ x : T \bullet A_1 \llbracket ns_1 \cup \{x\} \mid cs \mid ns_2 \rrbracket A_2)$$

provided $x \notin FV(A_2) \cup ns_1 \cup ns_2$ ❑

The new declared variable must be included in the partition of A_1 of the parallel composition. This is valid if the new variable x is not free in A_2 and if it is neither a member of ns_1 nor a member of ns_2.

An example of a *Circus* refinement law that has a direct correspondence with ZRC is the Alternation Introduction law.

Law C.140 (Alternation Introduction*).

$$w : [pre, post] \sqsubseteq_{\mathcal{A}} \mathbf{if}\ \rrbracket_i\, g_i \rightarrow\ w : [g_i \wedge pre, post]\ \mathbf{fi}$$

provided $pre \Rightarrow \bigvee_i g_i$ ❑

The proviso guarantees that the precondition of the specification ensures that at least one of the guards is valid. Additionally, we introduce each of the guards into the precondition of each of the guarded commands.

Another refinement law creates the link between alternations and guards. These two constructs differ in the cases where we have either no valid guards or more than one valid guard. We illustrate their differences in the binary case in the Table 4.1.

	$\mathbf{if}\ g_1 \rightarrow A_1 \rrbracket\ g_2 \rightarrow A_2\ \mathbf{fi}$	$g1\ \&\ A_2 \Box g_2\ \&\ A_2$
$g_1 \Leftrightarrow false \wedge g_2 \Leftrightarrow false$	$Chaos$	$Stop$
$g_1 \Leftrightarrow true \wedge g_2 \Leftrightarrow true$	$A_1 \sqcap A_2$	$A_1 \Box A_2$

Table 4.1: Alternation and Guards Different Behaviours

If neither g_1 nor g_2 are valid, the alternation diverges and the external choice between two actions guarded by g_1 and g_2 deadlocks. However, if both g_1 and g_2 are valid, the alternation internally chooses one of the branches; the external choice of the guarded action still offers the external choice to the environment. Apart from these two possibilities, both actions are the same.

Law C.141 (Alternation/Guarded Actions—interchange*).

$$\mathbf{if}\ g_1 \rightarrow\ A_1 \rrbracket\ g_2 \rightarrow A_2\ \mathbf{fi} = g_1\ \&\ A_1 \Box g_2\ \&\ A_2$$

provided $(g_1 \vee g_2) \wedge (g_1 \Rightarrow \neg\, g_2)$ ❑

The proviso guarantees that exactly one of the guards is valid.

4.4 Process Refinement

The laws for process refinement deal simultaneously with state and control behaviour. The first law applies to processes whose state components are partitioned in such a way that each partition has its own set of paragraphs. By way of illustration, we present the process P below.

$$\begin{array}{l}\textbf{process}\ P \mathrel{\widehat=} \textbf{begin state}\ State \mathrel{\widehat=} Q.State \land R.State \\ \qquad\qquad\qquad\qquad Q.PPar \land_{\Xi} R.State \\ \qquad\qquad\qquad\qquad R.PPar \land_{\Xi} Q.State \\ \qquad\qquad\qquad\qquad \bullet\ F(Q.Act, R.Act) \\ \qquad\qquad\quad\ \ \textbf{end}\end{array}$$

The state of the processes P is defined as a conjunction of two other schemas: $Q.State$ and $R.State$. Furthermore, the paragraphs in P are also partitioned in a way that the paragraphs in $Q.PPar$ do not change the components in $R.State$, since they are conjoined with $\Xi R.State$; in a similar way, the paragraphs in $R.PPar$ do not change the components in $Q.State$. Finally, the main action of P is defined as an action context F, which must also make sense as a function on processes, according to the *Circus* syntax (Appendix A).

The law presented below transforms such partitioned process into three processes: each of the first two includes a partition of the state and the corresponding paragraphs, and the third process, defined in the terms of the first two, has the same behaviour as the original one.

Law C.146 (Process splitting). Let qd and rd stand for the declarations of the processes Q and R, determined by $Q.State$, $Q.PPar$, and $Q.Act$, and $R.State$, $R.PPar$, and $R.Act$, respectively, and pd stands for the declaration of process P above.

$$pd = (qd\ \ rd\ \ \textbf{process}\ P \mathrel{\widehat=}\ F(Q, R))$$

provided $Q.PPar$ and $R.PPar$ are disjoint with respect to $R.State$ and $Q.State$. ❑

The second law applies to a process defined using the well-known Z promotion technique [107]. Using this family of laws, we may refine a specification using a free promotion to an indexed family of processes, each one representing an element of the local type. In what follows, the function $\mathbf{promote_2}$ extends the Z promotion technique to *Circus* actions. Firstly, as expected, we have that the promotion of schemas is as in Z.

$$\mathbf{promote_2}(SExp) \mathrel{\widehat=} \exists\, \Delta L.State \bullet SExp \land Promotion$$

$L.State$ stands for the local state, and $Promotion$ for the promotion schema.

The promotion of $Skip$, $Stop$, $Chaos$, and channels do not change them.

$$\begin{array}{l}\mathbf{promote_2}(A) \mathrel{\widehat=} A,\ \textit{for}\ A \in \{Skip, Stop, Chaos\} \\ \mathbf{promote_2}(c.e \to A) \mathrel{\widehat=} c.\mathbf{promote_2}(e) \to \mathbf{promote_2}(A)\end{array}$$

References to the local components have to become references to the corresponding com-

$$
\begin{array}{l}
\textbf{process}\ GAreas \mathrel{\widehat=} \\
\quad \textbf{begin}\ LState \mathrel{\widehat=} [\, id : AreaId;\ mode : Mode \,] \\
\quad \textbf{state}\ GState \mathrel{\widehat=} [\, f : AreaId \rightarrow LState \mid \forall\, a : AreaId \bullet (f\ a).id = a \,]
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\,Promotion\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta LState;\ \Delta GState;\ id? : Range \\
\hline
\theta LState = f\ id? \wedge f' = f \oplus \{ id? \mapsto \theta LState' \} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\quad LInit \mathrel{\widehat=} [\, LState' \mid mode' = automatic \,] \\
\quad GInit \mathrel{\widehat=} \forall\, id? : AreaId \bullet LInit \wedge Promotion \\
\quad LStart \mathrel{\widehat=} switchOn \rightarrow LInit \\
\quad GStart \mathrel{\widehat=} \lbrack\!\lbrack \{\!\mid switchOn \mid\!\} \rbrack\!\rbrack\ i : AreaId \bullet \\
\qquad\qquad\qquad \lbrack\!\lbrack\, \theta\,(f\ i) \,\rbrack\!\rbrack \bullet (\mathbf{promote_2}\ LStart)\,[id, id? := i, i] \\
\quad \bullet\ GStart \\
\quad \textbf{end}
\end{array}
$$

Figure 4.3: Process *GAreas*

ponent in the global state; all other references remain unchanged. An implicit parameter is a function f that maps indexes to instances of the local state. Another implicit parameter is the index i that identifies an instance of the local state in the global state.

$$
\begin{array}{ll}
\mathbf{promote_2}(x) \mathrel{\widehat=} (f\ i).x & \text{provided } x \text{ is a component of } L.State \\
\mathbf{promote_2}(x) \mathrel{\widehat=} x & \text{provided } x \text{ is not a component of } L.State
\end{array}
$$

The definitions of promotion for the other forms of prefix and actions are very similar; we need to promote every expression in the specification.

In Figure 4.3 we present a simplified version of a process that is part of our case study presented in Chapter 5. It consists of a fire control system that covers two separate physical areas. In this simplified version, each area has only an identification and a mode in which it is running. The process *GAreas* defines a controller for the areas covered by the system; the channel and type declarations are omitted.

The internal state of *GAreas* is declared as a function f that maps area identifications to local states *LState*. The local state of each area is composed of an identifier *id*, which is determined by the index of the area in f, and a *mode*. The *Promotion* schema is standard to Z; it relates an individual *LState* to the function f of areas. The global initialisation *GInit* is defined as the promotion of the local initialisations *LInit* of all areas. Similarly, *GStart* is declared as a parallel composition of the promotion of each local action *LStart*, which waits for the system to be switched on, and initialises the local state. The main action determines that, initially, *GAreas* behaves like *GStart*.

The behaviour of all the areas can be expressed in terms of the behaviour of each individual area. The process *LArea* presented in Figure 4.4 is parametrised by an identifier *id*; it represents the behaviour of a single area. Each *LArea* has a component that indicates

$$
\begin{array}{l}
\textbf{process } LArea \mathrel{\widehat=} (id : AreaId \bullet \textbf{begin state } LState \mathrel{\widehat=} [\, mode : Mode \,] \\
\qquad\qquad\qquad\qquad\qquad\qquad LInit \mathrel{\widehat=} [\, LState' \mid mode' = automatic \,] \\
\qquad\qquad\qquad\qquad\qquad\qquad LStart \mathrel{\widehat=} switchOn \rightarrow LInit \\
\qquad\qquad\qquad\qquad\qquad\qquad \bullet\ LStart \textbf{ end}) \\
\textbf{process } GAreas \mathrel{\widehat=} \llbracket \{\!|\, switchOn \,|\!\} \rrbracket\ id : AreaId \bullet LArea(id)
\end{array}
$$

Figure 4.4: Process *GAreas* Refined

in which *mode* it is running. This component is initialised to *automatic* by the operation *LInit*. Initially, *LArea* behaves like action *LStart*. The global behaviour *GAreas* can be rewritten as a parallel composition of all areas.

Law C.147 presented in Appendix C can be used to make the refinement of process *GAreas*. This law applies to processes containing a local and a global state *LState* and *GState*, local paragraphs that do not affect the global state, a promotion schema, and global paragraphs expressed in terms of the promotion of local paragraphs to the global state using iterated parallel operator. The results of this application are two processes: a local process *L* parametrised by an identifier *id* and a global process *G* defined as an iterated parallel composition of local processes.

4.5 Soundness of the Refinement Laws

The aim of our work presented in the last section is to provide a basis for a theorem prover for *Circus*. This theorem prover will support the development of *Circus* programs by providing a library of refinement laws which have been mechanically proved. However, before doing the mechanical proof, we have done the proofs by hand. We conclude this chapter by presenting three out of over a hundred proofs we have done. The first one, the external choice unit law, explicitly shows why we have chosen *Stop* to leave the state loose; the second one, the parallel composition deadlocked 1 law, illustrates our approach in the proofs involving parallel composition; we conclude this section by presenting the proof of a derived law, the prefix/external choice—distribution law.

External choice has a unit action, *Stop*. This fact is expressed by the Law C.114 presented below.

Law C.114 $Stop \,\Box\, A = A$

Before presenting the proof of this refinement law, we present two auxiliary lemmas that are used in the proof. The first one gives the conditions on which *Stop* diverges.

Lemma 4.1 $Stop^f_f = \neg\, okay \wedge tr \leq tr'$

Since the precondition of *Stop* is *true*, as we would expect, *Stop* only diverges if its predecessor has done so and, in this case, only guarantees that the trace history is not forgotten. The next lemma tells us the effects of *Stop* when it does not diverge.

Lemma 4.2 $Stop_f^t = \boldsymbol{CSP1}(tr' = tr \land wait')$

From the definition of **CSP1**, if the predecessor diverges, *Stop* only guarantees that the trace history is not forgotten; otherwise, it does not change the trace and waits indefinitely.

We start our proof by applying the definition of external choice.

$$
\begin{array}{ll}
Stop \;\Box\; A & \\
= R \left(\begin{array}{l} (\neg\, Stop_f^f \land \neg\, A_f^f) \\ \vdash \\ ((Stop_f^t \land A_f^t) \lhd tr' = tr \land wait' \rhd (Stop_f^t \lor A_f^t)) \end{array} \right) & \text{[External choice]}
\end{array}
$$

Next, we use Lemmas 4.1 and 4.2 to transform $Stop_f^f$ and $Stop_f^t$, respectively.

$$
= R \left(\begin{array}{l} (\neg\, (\neg\, okay \land tr \leq tr') \land \neg\, A_f^f) \\ \vdash \\ \left(\begin{array}{l} (\mathbf{CSP1}(tr' = tr \land wait') \land A_f^t) \\ \lhd tr' = tr \land wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t) \end{array} \right) \end{array} \right) \quad \text{[Lemmas 4.1 and 4.2]}
$$

The direct application of simple predicate calculus gives us the following result.

$$
= R \left(\begin{array}{l} (\neg\, ((\neg\, okay \land tr \leq tr') \lor A_f^f)) \\ \vdash \\ \left(\begin{array}{l} (\mathbf{CSP1}(tr' = tr \land wait') \land A_f^t) \\ \lhd tr' = tr \land wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t) \end{array} \right) \end{array} \right) \quad \text{[Predicate calculus]}
$$

The predicate A_f^f is a notation that corresponds to the substitution of $okay'$ and *wait* in A; however, the predicate $\neg\, okay \land tr \leq tr'$ does not mention either of these variables. Therefore, we may expand the substitution; this leaves us with the definition of **CSP1**.

$$
= R \left(\begin{array}{l} (\mathbf{CSP1}(A))_f^f \\ \vdash \\ \left(\begin{array}{l} (\mathbf{CSP1}(tr' = tr \land wait') \land A_f^t) \\ \lhd tr' = tr \land wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t) \end{array} \right) \end{array} \right) \quad \text{[Substitution and } \mathbf{CSP1}\text{]}
$$

Theorem 3.3 tells us that every *Circus* action is a **CSP1** process; therefore, the application of **CSP1** to A can be removed.

$$
= R \left(A_f^f \vdash \left(\begin{array}{l} (\mathbf{CSP1}(tr' = tr \land wait') \land A_f^t) \\ \lhd tr' = tr \land wait' \rhd \\ (\mathbf{CSP1}(tr' = tr \land wait') \lor A_f^t) \end{array} \right) \right) \quad \text{[Theorem 3.3]}
$$

Next, by expanding the definition of **CSP1**, we get the following disjunction.

$$
= R \left(A_f^f \vdash \left(\begin{array}{l} (((tr' = tr \land wait') \lor (\neg\, okay \land tr \leq tr')) \land A_f^t) \\ \lhd tr' = tr \land wait' \rhd \\ (((tr' = tr \land wait') \lor (\neg\, okay \land tr \leq tr')) \lor A_f^t) \end{array} \right) \right) \quad [\mathbf{CSP1}]
$$

The simple expansion of designs shows us that *okay* cannot be *false* in the postcondition;

hence, the predicate $\neg\, okay \wedge tr \leq tr'$ is *false*. This leaves us with the following reactive design.

$$= R\left(A^f_f \vdash \left(\begin{array}{l} (tr' = tr \wedge wait' \wedge A^t_f) \\ \lhd tr' = tr \wedge wait' \rhd \\ ((tr' = tr \wedge wait') \vee A^t_f) \end{array} \right) \right) \quad \text{[Design and Predicate calculus]}$$

At this point, we are able to contemplate our decision on the semantics of *Stop*. The next step in our proof is to remove the disjunction of the right-hand side of the condition and leave just the predicate A^t_f; this can be done because the expression $tr' = tr \wedge wait'$ is *false*. The condition comes direct from our definition of external choice, in which, as explained in Section 3.1, state changes have no direct consequence. If we had chosen state changes to decide the choice, this would be expressed by including the predicate $v' = v$ in the condition of the choice. If this were the case, then *Stop* would also have to leave the state unchanged. However, this is not the case, and hence, in order to go ahead with our proof, it is clear that *Stop* cannot restrict the state to be kept unchanged.

$$= R(A^f_f \vdash ((tr' = tr \wedge wait' \wedge A^t_f) \vee A^t_f)) \quad \text{[Conditional and Predicate calculus]}$$
$$= R(\neg\, A^f_f \vdash A^t_f) \quad \text{[Predicate calculus]}$$

Since, every *Circus* action is **CSP1-CSP3** healthy, the application of Theorem 3.2, concludes this proof.

$$= A \quad \text{[Theorem 3.2]}$$

❑

The next law states that the parallel composition of two actions that are willing to synchronise on some event in the synchronisation set, but have no common initial communications, deadlocks. Furthermore, if one of the processes is already deadlocked and the other is waiting to synchronise on some event, the parallel composition also deadlocks. The proof of Law C.92 is done in two parts: the left-hand side equality and the right-hand side equality. In what follows, we only present the proof of the first one. The proof of the second one is pretty similar and omitted here.

Law C.92 (Parallel composition Deadlocked 1*).

$$(c_1 \rightarrow A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \rightarrow A_2) = Stop = Stop \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \rightarrow A_2)$$

provided

➭ $c_1 \neq c_2$

➭ $\{c_1, c_2\} \subseteq cs$ ❑

As we did for the proof previously presented, we first discuss the lemmas that are used in the proof. The proofs of both lemmas are rather long; in what follows, we present them and give the intuition behind their proofs.

Lemma 4.3 presented below guarantees that, provided the previous action did not diverge, divergence is not feasible in this parallel composition. Intuitively, the prefixed action $c \to A$ may diverge, but the event c will be in the final trace, because **R1** guarantees that, even in divergence, the trace history is not forgotten. For this reason, since both channels are different and members of the synchronisation channel set, it is not possible to have two traces $1.tr'$ and $2.tr'$, where the first one is a trace of the left-hand side prefix, the second one is a trace of the right-hand side prefix, and they are equal, modulo the synchronisation set cs.

Lemma 4.3

$$\left(okay \wedge \neg\ \exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} ((c_1 \to A_1)^f_f;\ 1.tr' = tr) \\ \wedge\ ((c_2 \to A_2)_f;\ 2.tr' = tr) \\ \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array} \right) \right) = okay$$

provided $c_1 \neq c_2$ and $\{c_1, c_2\} \subseteq cs$.

The next lemma gives us the result of executing both actions independently and merging their behaviours, provided the previous action did not diverge.

Lemma 4.4

$$\left(okay \wedge \left(\left(\begin{array}{l} ((c_1 \to A_1)^t_f;\ U1) \\ \wedge\ ((c_2 \to A_2)^t_f;\ U2) \end{array} \right)_{+\{v,tr\}} ;\ M_{\|_{cs}} \right) \right) = okay \wedge tr' = tr \wedge wait'$$

provided $c_1 \neq c_2$ and $\{c_1, c_2\} \subseteq cs$.

The only behaviour that remains, if we expand all the possible behaviours of this execution followed by the merge, is the one that states that the trace is left unchanged and it is indefinitely waiting. This holds because, from the definition of $M_{\|_{cs}}$, the only possibilities that are considered are those in which the traces are equal modulo cs. However, in their first progress, both actions will already have a different trace. Therefore, the traces of both actions are the same only while they are waiting to synchronise.

We start our proof by expanding the definition of the parallel composition.

$$\begin{array}{ll} (c_1 \to A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \to A_2) & \\ = \mathbf{R} \left(\begin{array}{l} \neg\ \exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} ((c_1 \to A_1)^f_f;\ 1.tr' = tr) \\ \wedge\ ((c_2 \to A_2)_f;\ 2.tr' = tr) \\ \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array} \right) \\ \wedge \neg\ \exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} ((c_1 \to A_1)_f;\ 1.tr' = tr) \\ \wedge\ ((c_2 \to A_2)^f_f;\ 2.tr' = tr) \\ \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array} \right) \\ \vdash \\ (((c_1 \to A_1)^t_f;\ U1) \wedge ((c_2 \to A_2)^t_f;\ U2))_{+\{v,tr\}};\ M_{\|_{cs}} \end{array} \right) & [\text{Parallel}] \end{array}$$

Next, we apply the definition of a design and some trivial predicate calculus as follows.

[Designs and predicate calculus]

$$= \mathbf{R}\left(\begin{array}{l}\left(\begin{array}{l} okay \wedge \neg\, \exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} ((c_1 \rightarrow A_1)^f_f;\ 1.tr' = tr) \\ \wedge\ ((c_2 \rightarrow A_2)_f;\ 2.tr' = tr) \\ \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array}\right) \\ \qquad \wedge \neg\, \exists\, 1.tr', 2.tr' \bullet \left(\begin{array}{l} ((c_1 \rightarrow A_1)_f;\ 1.tr' = tr) \\ \wedge\ ((c_2 \rightarrow A_2)^f_f;\ 2.tr' = tr) \\ \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \end{array}\right) \end{array}\right) \Rightarrow \\ \quad \left(\begin{array}{l} okay \wedge okay' \\ \wedge\ (((c_1 \rightarrow A_1)^t_f;\ U1) \wedge ((c_2 \rightarrow A_2)^t_f;\ U2))_{+\{v,tr\}};\ M_{\|_{cs}} \end{array}\right)\end{array}\right)$$

The application of the Lemma 4.3 twice removes the existential quantification in the left-hand side of the implication.

$$\mathbf{R}\left(\begin{array}{l} okay \Rightarrow \\ \quad \left(\begin{array}{l} okay \wedge okay' \\ \wedge \left(\begin{array}{l} ((c_1 \rightarrow A_1)^t_f;\ U1) \\ \wedge\ ((c_2 \rightarrow A_2)^t_f;\ U2) \end{array}\right)_{+\{v,tr\}};\ M_{\|_{cs}} \end{array}\right)\end{array}\right) \qquad \text{[Lemma 4.3]}$$

And the application of the Lemma 4.4 gives us the actual postcondition of the design.

$$= \mathbf{R}(okay \Rightarrow okay \wedge okay' \wedge tr' = tr \wedge wait') \qquad \text{[Lemma 4.4]}$$

Finally, we apply the definition of designs again and use some predicate calculus in order to get a reactive design, which is in fact, the definition of *Stop* itself.

$$= \mathbf{R}(true \vdash tr' = tr \wedge wait') \qquad \text{[Predicate calculus and Designs]}$$

$$= Stop \qquad [Stop]$$

❑

Before concluding this section, we present the proof of a derived law. In such proofs, we do not need to expand any of the constructors' definitions, but simply use other refinement laws. The next law states that we may distribute prefix over an external choice of guarded actions, provided exactly one of the guards is valid.

Law C.104

$$c \rightarrow ((g_1 \,\&\, A_1) \,\Box\, (g_2 \,\&\, A_2)) = (g_1 \,\&\, c \rightarrow A_1) \,\Box\, (g_2 \,\&\, c \rightarrow A_2)$$

provided

➩ $\vee\, i \bullet g_i$

➩ $\forall\, i, j \mid i \neq j \bullet \neg\, (g_i \wedge g_j)$ (guards are mutually exclusive). ❑

The proof of this law is done by case analysis on the guards; since they are mutually exclusive, and at least one of them is *true*, we are left with two cases, either g_1 is *true* and

g_2 is *false*, or vice-versa. We present below the proof of the first case. First, we introduce the *Skip* using the sequence unit law, and transform it into a *true* assumption using the assumption unit law.

$$c \rightarrow ((g_1 \ \& \ A_1) \Box (g_2 \ \& \ A_2))$$
$$= \{true\};\ c \rightarrow ((g_1 \ \& \ A_1) \Box (g_2 \ \& \ A_2)) \qquad \text{[Laws C.132 and C.55]}$$

Our assumption in this part of the proof is that g_1 is *true* and g_2 is *false*; the next step can be done by simple predicate calculus.

$$= \{g_1 \wedge \neg\, g_2\};\ c \rightarrow ((g_1 \ \& \ A_1) \Box (g_2 \ \& \ A_2)) \qquad \text{[Predicate calculus]}$$

The distribution laws for assumption allows us to distribute it to each of the branches of the choice.

$$= \{g_1 \wedge \neg\, g_2\};\ c \rightarrow \left(\begin{array}{l} \{g_1 \wedge \neg\, g_2\};\ g_1 \ \& \ A_1 \\ \Box\ \{g_1 \wedge \neg\, g_2\};\ g_2 \ \& \ A_2 \end{array} \right) \qquad \text{[Laws C.42 and C.37]}$$

Since the assumption validates the guard g_1, we can eliminate this guard in the first branch of the external choice. Furthermore, from Law C.33 we have that the second branch is deadlocked, because the assumption negates the guard.

$$= \{g_1 \wedge \neg\, g_2\};\ c \rightarrow \left(\begin{array}{l} \{g_1 \wedge \neg\, g_2\};\ A_1 \\ \Box\ \{g_1 \wedge \neg\, g_2\};\ Stop \end{array} \right) \qquad \text{[Laws C.32 and C.33]}$$

Using the distribution laws once again, we move the assumption back to its original point.

$$= \{g_1 \wedge \neg\, g_2\};\ c \rightarrow (A_1 \Box\ Stop) \qquad \text{[Laws C.42 and C.37]}$$

The application of the unit law for external choice, twice, allows us to move the prefix to the first branch of the choice.

$$= \{g_1 \wedge \neg\, g_2\};\ ((c \rightarrow A_1) \Box\ Stop) \qquad \text{[Law C.114]}$$

Once again, we distribute the assumption over the external choice.

$$= (\{g_1 \wedge \neg\, g_2\};\ c \rightarrow A_1) \Box (\{g_1 \wedge \neg\, g_2\};\ Stop) \qquad \text{[Law C.37]}$$

And, in the same way we did to remove the guards, we use the Laws C.32 and C.33, but in this time, we re-introduce the guards.

$$\begin{array}{l} = \{g_1 \wedge \neg\, g_2\};\ (g_1 \ \& \ c \rightarrow A_1) \\ \quad \Box\ \{g_1 \wedge \neg\, g_2\};\ (g_2 \ \& \ c \rightarrow A_2) \end{array} \qquad \text{[Laws C.32 and C.33]}$$

Finally, in the same way we did in the start of this proof, we can remove the assumptions because we are assuming that g_1 is *true* and g_2 is *false*, and a *true* assumption is the same as *Skip*, which is the unit for sequence.

$$= (g_1 \ \& \ c \rightarrow A_1) \Box (g_2 \ \& \ c \rightarrow A_2) \qquad \text{[Predicate calculus and Laws C.55 and C.132]}$$

❑

Some of the refinement laws we propose in this thesis were also proved in this way: using other refinement laws. Following the approaches presented in this section, among others, we have proved over ninety percent of the refinement laws proposed here. The remaining proofs, although rather long, are not challenging; they are left as future work.

4.6 Final Considerations

Refinement plays a major role in the UTP, which is the theoretical basis for *Circus*. In the UTP, it is expressed as an implication: an implementation P satisfies a specification S if, and only if, $[P \Rightarrow S]$. In *Circus*, this definition of refinement is used for the most basic notion of refinement: action refinement.

For processes, since they encapsulate their state, we have that process refinement is defined in terms of action refinement between the main actions of the abstract and the concrete process, but we hide the state components of the processes as if they were declared in a local variable block; we are left with a state space containing only the UTP observational variables $okay$, $wait$, tr, and ref. For data refinement, the standard simulation techniques used in Z are adopted to handle processes and actions. However, since actions are total, their definitions differ slightly from the usual definitions: no applicability requirement concerning preconditions exists. Furthermore, since state initialisation must be explicitly included in the main action, no conditions is imposed on the initialisation.

The refinement strategy for *Circus* presented in [27], and discussed in this chapter, is based on laws of simulation, and action and process refinement. In this chapter, we have presented further laws of simulation and refinement. The need for these laws was raised during the development of the case study presented in Chapter 5, which is the first industrial application of the refinement technique. Some of the new laws could be proved using previously defined laws; these proofs are also presented in this thesis. The corrections of some of the laws from [27] were also discussed in this chapter. However, our work has also revealed that the following law proposed in [27] was not valid.

Law A.19 (Parallel composition Introduction: Sequence 2).

$$A_1;\ A_2(x) = (c!x \rightarrow A_1 \llbracket \overline{wrtV(A_2)} \mid \{\!|\ c\ |\!\} \mid wrtV(A_2) \rrbracket\ c?y \rightarrow A_2(y)) \setminus \{\!|\ c\ |\!\}$$

provided

- $wrtV(A_1) \cap usedV(A_2) = \varnothing$
- $c \notin usedC(A_1) \cup usedC(A_2)$
- $y \notin FV(A_2)$ ❑

After the hidden synchronisation in c, the action in the right-hand side of the law behaves as an interleaving between A_1 and A_2. This is a direct consequence of the fact that c is not used by any of these actions. However, the action in the left-hand side of the law is a sequential composition; the law is not valid.

This chapter also presented Theorem 4.3 that allows most of the laws from ZRC to be used in the *Circus* refinement calculus. This is a very important result since *Circus* specifications may contain Z schemas and specification statements.

The case studies that have been carried out on the *Circus* refinement calculus give us confidence that the set of laws that is presented here is appropriate for useful applications. We do not seek a completeness result; in fact, we know that our laws are not complete, because we only consider forwards simulation and leave backwards simulation as future work. In the future, we plan to provide an algebraic semantics for *Circus*, define a normal form, and establish that we have a set of laws that is enough to reduce any terminating *Circus* program to its normal form. The laws of an algebraic semantics, however, are equalities; in this work, we are concerned with the practicalities of refinement.

Laws of programming have been of interest for a very long time [53]. Laws for imperative, functional [14], and object-oriented languages [16] are available in the literature. Our laws are more closely related to those presented in [81] to provide an algebraic semantics for terminating occam programs. Our laws, however, are aimed at supporting program development; we focus on novel laws that relate constructs to manipulate data with constructs to specify behaviour. We present equalities and also refinement laws.

The vast majority of the laws presented in this thesis were proved; in this chapter, we illustrated these proofs and pointed out the interesting aspects raised. For instance, the proof of Law C.114 makes clear the design options we make regarding the aspects that would resolve a choice; for us, state change does not resolve a choice and, as a consequence, *Stop* leaves the state loose in its postcondition.

Some other laws proposed in this work were proved in terms of other refinement laws. Although not strictly needed, these laws provide shortcuts for the users of our method, shortening the development of programs. This result could also be obtained with the use of tactics of refinement, like those presented in [74, 70, 72]; we leave this as future work.

Throughout this thesis, we consider *Circus* specifications that do not mention any of the UTP variables. Some of our laws (e.g, Law C.45) would not be valid if this were not the case. An investigation on the advantages and consequences of *Circus* specifications that do mention the UTP variables is an interesting piece of future research.

The mechanisation of these proofs is a hard task that will provide *Circus* with a theorem prover that can be used in the development process shown in this thesis; it is left as future work as well.

Chapter 5

Case Study

In this chapter we present a case study on the *Circus* refinement calculus. The case study is a safety-critical fire control system, that is described in Section 5.1. In Section 5.2, we describe the types and channels used within the system, some axiomatic definitions that are used throughout its specification and design, and an abstract specification for the fire control system. In Section 5.3, this specification is refined to a concrete design using the refinement strategy presented in Chapter 4. Finally, in Section 5.4 we present some conclusions on the case study. The material in this chapter was published in [75, 76].

5.1 System Description

Our case study consists of a fire control system, which was implemented by Wormald Ltd. The system monitors fire detections in six distributed zones: four of these zones are distributed into two different areas (two zones for each area) and the two remaining zones are for fire detection only (see Figure 5.1). A fire detection in one or more zones may lead to a gas discharge in the area that includes the zone in which the fire was detected. If, however, a fire is detected in one of the zones used only for fire detection, no gas is discharged. In both cases, the detection of a fire is indicated in a display panel, which also indicates whether the system is on or off, there are system faults, the alarm has been silenced, the actuators of the system need to be replaced, or any gas discharge has happened.

The system can be in one of three modes: manual, automatic, or disabled. In manual mode, an alarm sounds when a fire is detected, and the corresponding detection lamp is

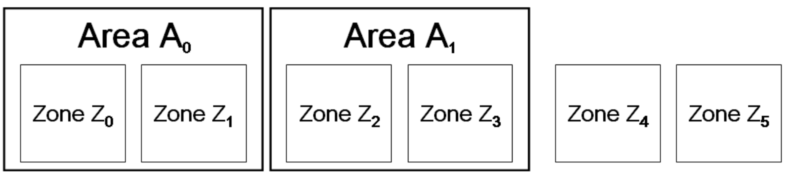


Figure 5.1: Zones and Areas in the Fire Control System

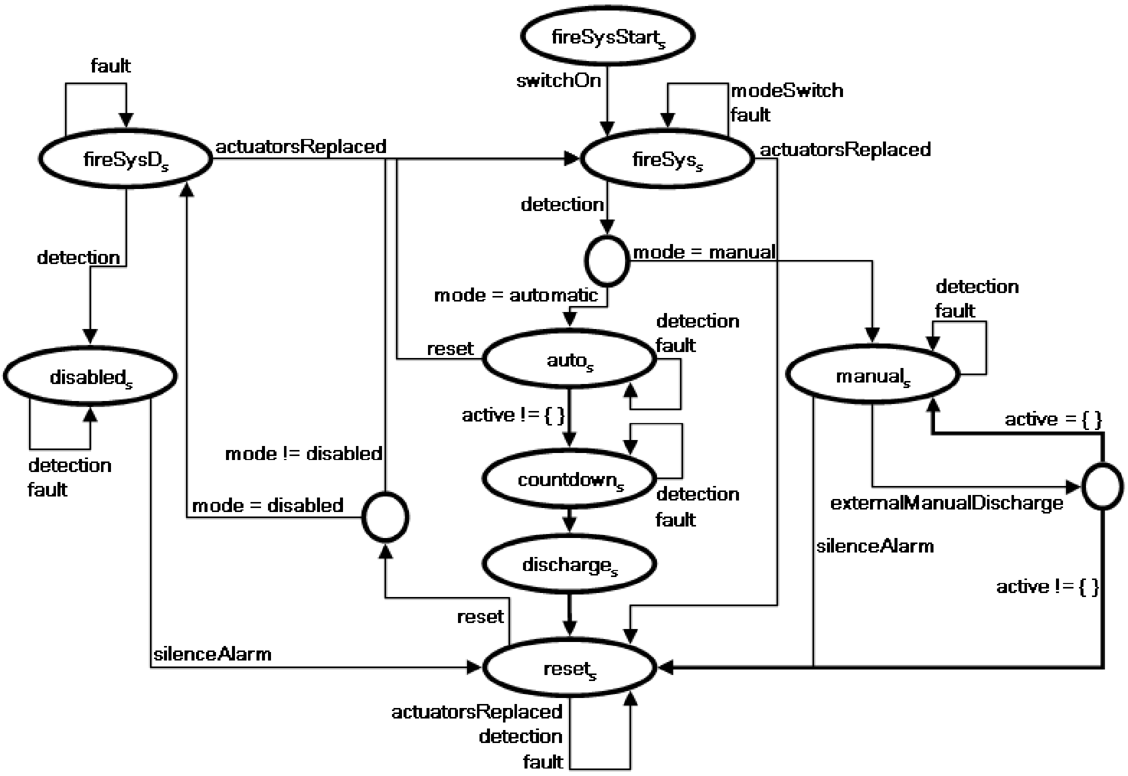


Figure 5.2: Fire Control System State Machine

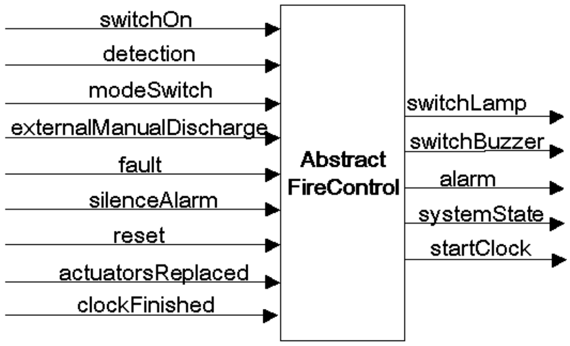
lit on the display. The alarm can be silenced, and, when the reset button is pressed, the system returns to normal. In manual mode, gas discharge is manually initiated.

In automatic mode, a fire detection is followed by the alarm being sounded; however, if a fire is detected in a second zone of the same area, a second stage alarm is sounded, and a countdown starts. When the countdown finishes, the gas is discharged and the circuit fault lamp is illuminated in the display; the system mode is switched to disabled.

In disabled mode, the system can only have the actuators replaced, identify relevant faults within the system, and be reset. The system is back to its normal mode after the actuators are replaced and the reset button is pressed.

Some further requirements should also be satisfied: the system must be started with a *switchOn* event, and afterwards the system on lamp should be illuminated; the system mode can be switched between manual and automatic mode, provided no detection happens. Also, when the system is reset, all fire detection lamps must be switched off; if a gas discharge occurred, the actuators need to be replaced, and the system mode is switched to automatic. Following a fire detection, the corresponding lamp must be lit. After a gas discharge, no subsequent discharge may happen before the actuators are replaced.

To summarise, the system may be in one of the states presented in Figure 5.2. Initially, the system is in the $fireSysStart_s$ state. After being switched on, its state is changed to $fireSys_s$; in this state, a fire detection results in the state being changed to $manual_s$ or



channel *switchOn*, *silenceAlarm*, *reset*
channel *actuatorsReplaced*, *startClock*, *clockFinished*
channel *detection* : *ZoneId*
channel *modeSwitch* : *SwitchMode*
channel *externalManualDischarge* : $\mathbb{P}$ *AreaId*
channel *fault* : *FaultId*
channel *alarm* : *AlarmStage*
channel $[T]$*switchLamp* : $T \times$ *OnOff*
channel *switchBuzzer* : *OnOff*
channel *systemState* : *SystemState*

Figure 5.3: System External Channels

$auto_s$ depending on the system mode. In $countdown_s$, it is waiting for the clock to finish the countdown. During gas discharge, the system is in the $discharge_s$ state. After the gas discharge happens, the state is changed to $reset_s$ and the system mode is automatically set to disabled. In the $reset_s$ state, the system is waiting to be reset. If the actuators are replaced, the system continues in the $reset_s$ state, but its mode is changed to automatic. If the system is reset in a disabled mode, its state is changed to $fireSysD_s$; otherwise, it goes back to the $fireSys_s$ state. A fire detection in the $fireSysD_s$ state results in the system state being changed to $disabled_s$. Finally, if the alarm is silenced in the $disabled_s$ state, the system goes back to the $reset_s$ state.

The external channels of the fire control system are presented in Figure 5.3. Fire detection is indicated through the channel *detection*, which inputs the zone where it happened. The system mode can be manually switched using the channel *modeSwitch*. In manual mode, when the conditions that lead to a gas discharge are met, gas can be manually discharged using the channel *externalManualDischarge*. Faults are reported to the system through the channel *fault*. The channel *alarm* can be used to sound the alarm, which can be silenced through *silenceAlarm*. Channel *reset* resets the system. The channel *actuatorsReplaced* indicates that the actuators have been replaced. The system indicates that a lamp must be switched using the generic channel *switchLamp*; it provides the type of lamp and the new lamp mode. The buzzer is controlled using

$$
\begin{aligned}
&AreaId ::= A_0 \mid A_1\\
&ZoneId ::= Z_0 \mid Z_1 \mid Z_2 \mid Z_3 \mid Z_4 \mid Z_5\\
&Mode ::= automatic \mid manual \mid disabled\\
&SwitchMode == Mode \setminus \{disabled\}\\
&OnOff ::= on \mid off\\
&AlarmStage ::= alarmOff \mid firstStage \mid secondStage\\
&LampId ::= zoneFaultLamp \mid earthFaultLamp \mid sounderLineFaultLamp\\
&\qquad \mid powerFaultLamp \mid systemOnLamp \mid isolateRemoteSignalLamp\\
&\qquad \mid actuatorLineFaultLamp \mid circuitFaultLamp \mid alarmSilencedLamp\\
&FaultId ::= zoneFault \mid earthFault \mid sounderLineFault \mid powerFault\\
&\qquad \mid isolateRemoteSignal \mid actuatorLineFault\\
&SystemState ::= fireSysStart_s \mid fireSys_s \mid fireSysD_s \mid auto_s\\
&\qquad \mid countdown_s \mid discharge_s \mid reset_s \mid manual_s \mid disabled_s
\end{aligned}
$$

Figure 5.4: System Types

the channel *switchBuzzer*. After each state change, the system reports its current state using the channel *systemState*. The fire control system may request a clock to execute the countdown using the channel *startClock*; the clock indicates that the countdown is finished using the channel *clockFinished*.

The display is composed of the lamps and the buzzer. The lamps can be of three different types; however, the three types of lamps are instances of the same generic process *GenericLamp*, which has a component *status* of a type *OnOff* that contains two values: *on* and *off*. Initially, all the lamps are switched *off*; they can be switched *on* using an appropriate instance of channel *switchLamp*.

5.2 Abstract Fire Control System

The basic types used within the system are presented in Figure 5.4. The areas and zones are identified by the types *AreaId* and *ZoneId*; the system modes are represented by the type *Mode*; the type *SwitchMode*, is a subset of the type *Mode*. All the lamps and the buzzer of the display can be either *on* or *off*, which are represented by the type *OnOff*. The alarm states are represented by the type *AlarmStage*. The type *LampId* contains identifiers for all the lamps in the system's display. Faults are represented by the type *FaultId*. Finally, the system can be in one of the states of the type *SystemState*.

Process *AbstractFireControl* formalises the requirements previously described. In this Chapter we omit some formal definitions for the sake of conciseness; they can be found in [71]. The abstract state is defined by the Z schema named *AbstractFireControlState* presented below. *AbstractFireControlState* contains five components: *mode* indicates the mode in which the fire control is running; *controlledZones* is a total function that maps the areas to a set that contains their controlled zones; *activeZones* maps the areas to the zones in which a fire detection has occurred; *discharge* indicates in which areas a gas discharge happened; finally, *active* contains the active areas identifications.

process *AbstractFireControl* $\widehat{=}$ **begin**
state

$$\begin{array}{|l}
\textit{AbstractFireControlState} \\
\hline
mode : Mode \\
controlledZones, activeZones : AreaId \rightarrow \mathbb{P}\ ZoneId \\
discharge, active : \mathbb{P}\ AreaId \\
\hline
\forall a : AreaId \bullet \\
\quad (mode = manual) \Rightarrow a \in active \Leftrightarrow \#activeZones\ a \geq 1 \\
\quad \land (mode = automatic) \Rightarrow a \in active \Leftrightarrow \#activeZones\ a \geq 2 \\
\quad \land activeZones\ a \subseteq controlledZones\ a \\
\quad \land controlledZones\ a = getZones\ a \\
\hline
\end{array}$$

The state invariant determines that, if the system is running in *manual* mode (predicate $mode = manual$), an area is *active* if, and only if, some zone controlled by it is active. On the other hand, if the mode is *automatic*, an area is active if, and only if, there is more than one active zone controlled by it. Finally, for each area, its controlled zones are defined by the function *getZones*, whose definition we omit. In Z, $\#s$ is the cardinality of the set s.

Initially, the system is in *automatic* mode, there is no active zone, and no discharge occurred in any area. The state invariant guarantees that there is no active area.

$$\begin{array}{|l}
\textit{InitAbstractFireControl} \\
\hline
AbstractFireControlState' \\
\hline
mode' = automatic \land discharge' = \varnothing \\
activeZones' = \{a : AreaId \bullet a \mapsto \varnothing\} \\
\hline
\end{array}$$

Three operations are used to switch the system mode; they leave the other components unchanged. The first operation receives the new mode as argument.

$$\begin{array}{|l}
\textit{SwitchAbstractFireControlMode} \\
\hline
\Delta AbstractFireControlState;\ nm? : Mode \\
\hline
mode' = nm? \land activeZones' = activeZones \\
discharge' = discharge \\
\hline
\end{array}$$

SwitchAbstractFireControl2AutomaticMode and *SwitchAbstractFireControl2DisabledMode* do not receive arguments; they switch the mode to *automatic* and *disabled*, respectively.

The schema *AbstractActivateZone* receives a zone *nz*? and changes *activeZones* by including *nz*? in the set of active zones of the area that controls it; *active* may also be

State	Abstract FireControl	Concrete FireControl	Concrete Area
$fireSysStart_s$	$AbstractFireSysStart$	$FireSysStart$	$StartArea$
$fireSys_s$	$AbstractFireSys$	$FireSys$	$AreaCycle$
$manual_s$	$AbstractManual$	$Manual$	$ManualArea$
$auto_s$	$AbstractAuto$	$Auto$	$AutoArea$
$reset_s$	$AbstractReset$	$Reset$	$ResetArea$
$countdown_s$	$AbstractCountdown$	$Countdown$	$WaitingDischarge$
$discharge_s$	$AbstractDischarge$	$Discharge$	$WaitingDischarge$
$fireSysD_s$	$AbstractFireSysD$	$FireSysD$	$AreaD$
$disabled_s$	$AbstractDisabled$	$Disabled$	$DisabledArea$

Table 5.1: The System States and Corresponding Actions

changed to maintain the state invariant. All other state components are left unchanged.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;AbstractActivateZone\;}} \\
\Delta AbstractFireControlState;\ nz? : ZoneId \\
\hline
mode' = mode \land discharge' = discharge \\
activeZones' = activeZones \oplus \{a : AreaId \\
\qquad\qquad\qquad\qquad\qquad \mid nz? \in controlledZones\ a \\
\qquad\qquad\qquad\qquad\qquad \bullet\ a \mapsto activeZones\ a \cup \{nz?\}\}
\end{array}
$$

The schema *AbstractAutomaticDischarge* activates the discharge in the active areas; only *discharge* is changed. Finally, *AbstractManualDischarge* receives the areas in which the user wants to discharge the gas, but discharges only in those that are *active*.

All the other actions are defined using CSP operators. We have one action for each possible state within the system as described in Table 5.2.

The action *AbstractFireSysStart* starts by communicating the current system state. Then, it waits for the system to be switched *on* through channel *switchOn*, switches on the lamp *systemOnLamp*, initialises the system state and, finally, behaves like action *AbstractFireSys*.

$$
\begin{array}{l}
AbstractFireSysStart \mathrel{\widehat=} \\
\quad systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\qquad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\qquad\quad InitAbstractFireControl;\ AbstractFireSys
\end{array}
$$

In the action *AbstractFireSys*, after communicating the system state, the mode can be manually switched between *automatic* and *manual*. Furthermore, if any detection occurs, the zone in which the detection occurred is activated, the corresponding lamp is lit, the alarm sounds in *firstStage*, and then, the system behaves like *AbstractManual* or *AbstractAuto*, depending on the current system mode. If the actuators are replaced, the *circFaultLamp* is switched *off*, the system is set to *automatic* mode, and waits to be *reset*.

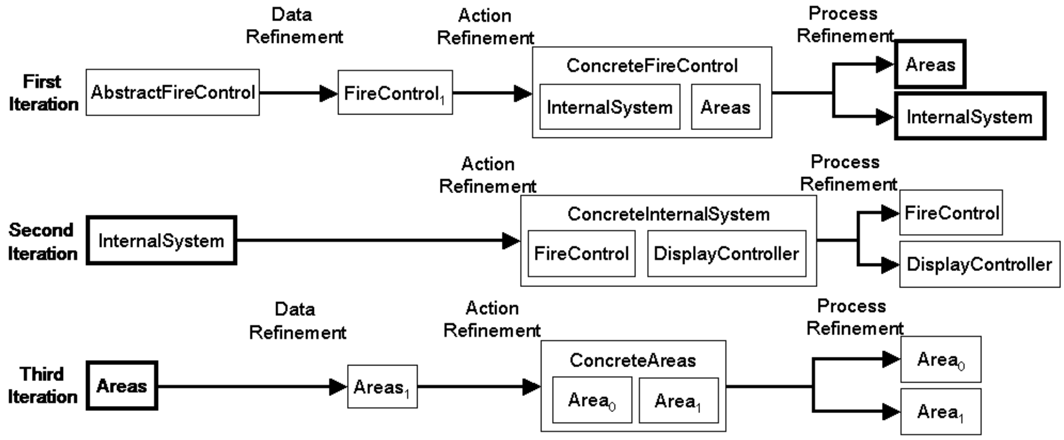


Figure 5.5: Refinement Strategy for the Fire Control System

Finally, if any *fault* is identified, the corresponding *switchLamp* is lit, and the buzzer is switched *on*.

$$
\begin{array}{l}
AbstractFireSys \mathrel{\widehat=} \\
\quad systemState!fireSys_s \rightarrow \\
\quad\quad modeSwitch?nm \rightarrow SwitchAbstractFireControlMode;\ AbstractFireSys \\
\quad\quad \Box\ detection?nz \rightarrow AbstractActivateZone;\ switchLamp[ZoneId].nz!on \rightarrow \\
\quad\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad\quad (mode = manual)\ \&\ AbstractManual \\
\quad\quad\quad\quad \Box\ (mode = automatic)\ \&\ AbstractAuto \\
\quad\quad \Box\ actuatorsReplaced \rightarrow switchLamp[LampId].circFaultLamp!off \rightarrow \\
\quad\quad\quad SwitchAbstractFireControl2AutomaticMode;\ AbstractReset \\
\quad\quad \Box\ fault?faultId \rightarrow switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow AbstractFireSys \\
\ldots
\end{array}
$$

The function *getLampId* maps fault identifications to their corresponding lamp in the display.

Throughout this chapter, we illustrate the refinement of the fire control system using these two actions only. For this reason, we omit the definitions of the remaining actions. All the definitions can be found in [71].

The main action of process *AbstractFireControl* is defined below.

$\bullet$ *AbstractFireSysStart* **end**

In the next section, we refine *AbstractFireControl* to a concrete distributed system.

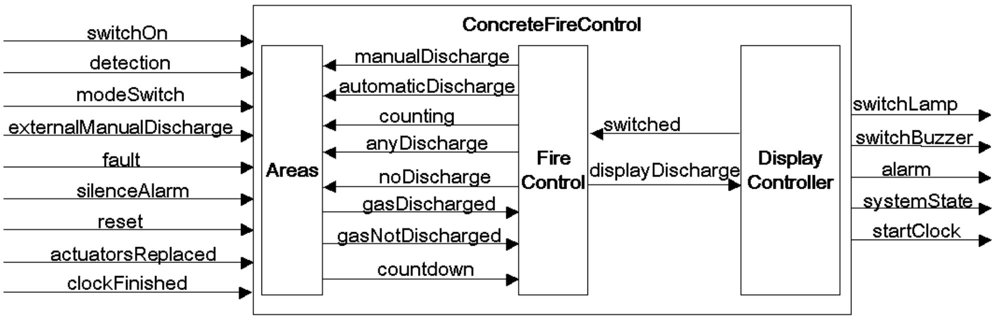


Figure 5.6: Concrete Fire Control

5.3 Refinement

The motivation for the fire control system refinement is the natural distribution arising from the physical locations of actuators, sensors, zones, and areas. Section 5.3.1 presents the target of our refinement, the concrete fire control system. In the following sections, we present the refinement steps summarised graphically in Figure 5.5.

In the first iteration, we split *AbstractFireControl* into two process. The first, *Areas*, models the areas of the system, and is split into two *Area* processes in parallel in the last iteration. The second, *InternalSystem*, is the core of the system, which is split into a display controller *DisplayController* and the system controller *FireControl* in the second iteration.

5.3.1 Concrete Fire Control System

The concrete fire control system has three components: the controller, the display, and the detection system. They communicate through the channels below.

$$
\begin{array}{ll}
\textbf{channel} & \mathit{displayDischarge}, \mathit{manualDischarge} : \mathbb{P}\,\mathit{AreaId} \\
\textbf{channel} & \mathit{switched}, \mathit{automaticDischarge}, \mathit{anyDischarge}, \\
& \mathit{noDischarge}, \mathit{countdown}, \mathit{counting} \\
\textbf{channel} & \mathit{gasDischarged}, \mathit{gasNotDischarged} : \mathit{AreaId}
\end{array}
$$

The controller indicates discharges to the display through *displayDischarge*. The display acknowledges this communication through channel *switched*. The detection process may request a *countdown* to the controller, if it is in *automatic* mode and the conditions for a gas discharge are met. The controller indicates that it started counting through *counting*. The controller requests gas discharges to the detection process through *manualDischarge* and *automaticDischarge*. Each of the areas in the detection process replies to the controller's request via channels *gasDischarged* or *gasNotDischarged*. After receiving all the answers from the areas, the controller indicates to all areas if any discharge has happened (*anyDischarge*) or not (*noDischarge*). In Figure 5.6, we summarise the internal communications of the concrete fire control system.

Controller The process *FireControl* is similar to the abstract specification. However, all the state components and events related to the areas and to the display are removed.

$$\begin{array}{l}\textbf{process } \mathit{FireControl} \mathrel{\widehat=} \textbf{begin state } \mathit{FireControlState} \mathrel{\widehat=} [\, \mathit{mode}_1 : \mathit{Mode} \,] \\ \quad \mathit{InitFireControl} \mathrel{\widehat=} [\, \mathit{FireControlState}' \mid \mathit{mode}_1' = \mathit{automatic} \,]\end{array}$$

The state of the concrete fire control contains only one component, mode_1, which indicates the mode in which the system is running. This mode is initialised to *automatic*; three operations can be used to switch it. The first one receives the new mode as argument.

$$\mathit{SwitchFireControlMode} \mathrel{\widehat=} [\, \Delta \mathit{FireControlState};\ \mathit{nm}? : \mathit{Mode} \mid \mathit{mode}_1 = \mathit{nm}? \,]$$

The second and third operations do not receive any argument; they simply switch the system mode to *automatic* or *disabled*.

The fire control system is responsible for communicating the current system state. After being switched *on*, the fire control initialises its state and behaves like action *FireSys*. Where a lamp was switched *on* in the abstract specification, an acknowledgment event *switched* is received from the display controller.

$$\begin{array}{l}\mathit{FireSysStart} \mathrel{\widehat=} \mathit{systemState}!\mathit{fireSysStart}_s \rightarrow \mathit{switchOn} \rightarrow \mathit{switched} \rightarrow \\ \qquad\qquad\qquad \mathit{InitFireControl};\ \mathit{FireSys}\end{array}$$

Similar to the abstract system, all the other actions corresponds to a possible state within the system as described in Table 5.2.

In action *FireSys*, after communicating the system state, the mode can be switched. Furthermore, if any detection occurs, the controller waits for a *switched* signal, sets the alarm to *firstStage*, and behaves like *Manual* or *Auto*, depending on the current system mode. Since the areas are the processes which have the area-zone information, following a *detection* communication, the zone activation is not part of the controller behaviour. If the actuators are replaced, the system is set to *automatic* mode, and waits to be *reset*. Finally, all the faults are ignored by this process, except that it waits for a *switched* signal from the display.

$$\begin{array}{l}\mathit{FireSys} \mathrel{\widehat=} \mathit{systemState}!\mathit{fireSys}_s \rightarrow \\ \qquad \left(\begin{array}{l}\mathit{modeSwitch}?\mathit{nm} \rightarrow \mathit{SwitchFireControlMode};\ \mathit{FireSys} \\ \Box\ \mathit{detection}?\mathit{nz} \rightarrow \mathit{switched} \rightarrow \mathit{alarm}!\mathit{firstStage} \rightarrow \\ \qquad (\mathit{mode}_1 = \mathit{manual})\ \&\ \mathit{Manual} \\ \qquad \Box\ (\mathit{mode}_1 = \mathit{automatic})\ \&\ \mathit{Auto} \\ \Box\ \mathit{actuatorsReplaced} \rightarrow \mathit{switched} \rightarrow \\ \qquad \mathit{SwitchFireControl2Auto};\ \mathit{Reset} \\ \Box\ \mathit{fault}?\mathit{faultId} \rightarrow \mathit{switched} \rightarrow \mathit{FireSys}\end{array}\right)\end{array}$$

As for the abstract system, we omit the definition of the remaining actions.

• *FireSysStart* **end**

The main action of process *FireControl* is *FireSysStart* presented above.

Display Controller This process models the display controller requests for the lamps to be switched *on* or *off* after the occurrence of the relevant events. It waits for the system to be switched *on*, switches the lamp *systemOnLamp on*, and indicates this to *FireControl* through *switched*. A gas discharge is indicated by *FireControl* to this process through *displayDischarge*. If the system is *reset*, the display switches *off* the buzzer and all the lamps, except the lamps *circFaultLamp* and *systemOnLamp*.

Areas The process *Area* is parametrised by the area identifier.

process $Area \mathrel{\widehat=} (id : AreaId \bullet$ **begin**

The state of an area is composed of the mode in which it is running, its controlled zones, the active zones in which a fire detection occurred, a boolean *discharge* that records whether a gas discharge has occurred or not, and a boolean *active* that records whether the area is willing to discharge gas or not.

state *AreaState*

$$
\begin{array}{|l}
mode : Mode \\
controlledZones, activeZones : \mathbb{P}\, ZoneId \\
discharge, active : Bool \\
\hline
controlledZones = getZones\ id \land activeZones \subseteq controlledZones \\
(mode = automatic) \Rightarrow active = true \Leftrightarrow \#activeZones \geq 2 \\
(mode = manual) \Rightarrow active = true \Leftrightarrow \#activeZones \geq 1
\end{array}
$$

The invariant establishes that the component *activeZones* is a subset of the controlled zones of this area, which is defined by *getZones*. If running in *automatic* mode, an area is active if, and only if, all controlled zones are *active*. On the other hand, if running in *manual* mode, an area is *active* if, and only if, any controlled zone is active.

Each area is initialised as follows: there is no active zone; no discharge occurred; and it is in *automatic* mode. The state invariant guarantees that it is not *active*.

InitArea

$$
\begin{array}{|l}
AreaState' \\
\hline
activeZones' = \varnothing \\
discharge' = false \\
mode' = automatic
\end{array}
$$

The schema *SwitchAreaMode* receives the new mode and sets the area mode. The schemas *SwitchArea2AutomaticMode* and *SwitchArea2DisabledMode* set the area *mode* to *automatic* and *disabled*. All other state components are left unchanged. A zone can be activated using the operation *ActivateZone*. If the given zone is controlled by the area, it is included in the *activeZones*.

Initially, an area synchronises in *switchOn*, initialises its state, and starts its cycle.

$$StartArea \mathrel{\widehat=} switchOn \rightarrow InitArea;\ AreaCycle$$

During its cycle, if the *actuatorsReplaced* event occurs, the mode is switched to *automatic* and the area waits to be *reset*. If the system mode is switched, so is the area mode. Finally, any detection may activate a zone, if it is controlled by this area; after this, the area behaves like either *AutoArea* or *ManualArea*, depending on its current mode.

$$\begin{array}{ll} AreaCycle \mathrel{\widehat=} & actuatorsReplaced \rightarrow SwitchArea2AutomaticMode; ResetArea \\ & \Box\ modeSwitch?nm \rightarrow SwitchAreaMode;\ AreaCycle \\ & \Box\ detection?nz \rightarrow ActivateZone; \\ & \quad (mode = automatic)\ \&\ AutoArea \\ & \quad \Box\ (mode = manual)\ \&\ ManualArea \\ \ldots\ \bullet\ StartArea\ \mathbf{end}) & \end{array}$$

The main action of the process *Area* is the action *StartArea*.

The process *ConcreteAreas* represents all the areas within the system. It is a parallel composition of all areas synchronising on the channel set Σ_{areas}.

$$\begin{array}{l} \mathbf{chanset}\ \Sigma_{areas} == \{\!|\ switchOn, reset, modeSwitch, detection, silenceAlarm, \\ \qquad\qquad actuatorsReplaced, automaticDischarge, manualDischarge, \\ \qquad\qquad anyDischarge, noDischarge, counting\ |\!\} \\ \mathbf{process}\ ConcreteAreas \mathrel{\widehat=}\ [\![\, \Sigma_{areas}\,]\!]\ id : AreaId \bullet Area(id) \end{array}$$

The internal system is defined as the parallel composition of the fire control *FireControl* and the display controller *DisplayController*. All the communications between them are hidden.

$$\begin{array}{l} \mathbf{chanset}\ DisplaySync == \{\!|\ displayDischarge, switched\ |\!\} \\ \mathbf{chanset}\ \Sigma_1 == \{\!|\ switchOn, reset, detection, displayDischarge, silenceAlarm, \\ \qquad\qquad actuatorsReplaced, fault\ |\!\} \\ \mathbf{process}\ ConcreteInternalSystem \mathrel{\widehat=} \\ \qquad FireControl\ [\![\, \Sigma_1\,]\!]\ DisplayController \setminus DisplaySync \end{array}$$

The concrete fire control is the parallel combination of *ConcreteInternalSystem* and *Areas*. Internal communications are again hidden.

$$\begin{array}{l} \mathbf{chanset}\ GasDischargeSync == \\ \qquad \{\!|\ manualDischarge, automaticDischarge, countdown, counting, \\ \qquad\quad gasDischarged, gasNotDischarged, anyDischarge, noDischarge\ |\!\} \\ \mathbf{chanset}\ \Sigma_2 == \{\!|\ switchOn, reset, detection, modeSwitch, silenceAlarm, \\ \qquad\qquad actuatorsReplaced\ |\!\} \cup GasDischargeSync \\ \mathbf{process}\ ConcreteFireControl \mathrel{\widehat=} \\ \qquad (ConcreteInternalSystem\ [\![\, \Sigma_2\,]\!]\ Areas) \setminus GasDischargeSync \end{array}$$

In the following sections, we prove that *AbstractFireControl* is refined by the process

ConcreteFireControl, or rather, $AbstractFireControl \sqsubseteq_{\mathcal{P}} ConcreteFireControl$.

5.3.2 First Iteration: splitting the *AbstractFireControl* into the internal controller and the areas processes

Data refinement In this step we make a data refinement in order to introduce a state component that is used by the areas. The new $mode_A$ component indicates the mode in which the areas are running. The process *AbstractFireControl* is refined to the process $FireControl_1$ presented below.

process $FireControl_1 \mathrel{\widehat=}$ **begin**
state

$$
\begin{array}{|l}
\textit{FireControlState}_1 \\
\hline
mode_1, mode_A : Mode \\
controlledZones_1, activeZones_1 : AreaId \rightarrow \mathbb{P}\, ZoneId \\
discharge_1, active_1 : \mathbb{P}\, AreaId \\
\hline
\forall a : AreaId \bullet \\
\quad (mode_1 = automatic) \Rightarrow a \in active_1 \Leftrightarrow \#activeZones_1\ a \geq 2 \\
\quad \land (mode_1 = manual) \Rightarrow a \in active_1 \Leftrightarrow \#activeZones_1\ a \geq 1 \\
\quad \land activeZones_1\ a \subseteq controlledZones_1\ a \\
\quad \land controlledZones_1\ a = getZones\ a \\
\hline
\end{array}
$$

The state $FireControlState_1$ is the same as that of *AbstractFireControl*, except that it includes an extra component $mode_A$. In order to prove that the $FireControl_1$ is a refinement of the *AbstractFireControl*, we have to prove that there exists a forwards simulation between the main actions of $FireControl_1$ and *AbstractFireControl*. The retrieve relation *RetrFireControl* relates each component in the *AbstractFireControlState* to one in $FireControlState_1$; it states that $mode_A$ is a duplicated record of *mode*.

$$
\begin{array}{|l}
\textit{RetrFireControl} \\
\hline
AbstractFireControlState;\ FireControlState_1 \\
\hline
mode_1 = mode \land mode_A = mode \land active_1 = active \\
controlledZones_1 = controlledZones \\
activeZones_1 = activeZones \land discharge_1 = discharge \\
\hline
\end{array}
$$

The laws of *Circus* establish that simulation distributes through the structure of an action. We refine each schema using Law C.4. In the concrete initialisation, the new state component $mode_A$ is initialised in *automatic* mode.

$$
\begin{array}{|l}
\textit{InitFireControl}_1 \\
\hline
FireControlState_1' \\
\hline
mode_1' = automatic \land mode_A' = automatic \land discharge_1' = \varnothing \\
activeZones_1' = \{a : AreaId \bullet a \mapsto \varnothing\} \\
\hline
\end{array}
$$

The following lemma states that this is actually simulated by the abstract initialisation.

Lemma 5.1 $InitAbstractFireControl \preceq InitFireControl_1$

Proof. The application of Law C.4 raises two proof obligations. The first one concerns the preconditions of both schemas.

$$\begin{array}{l}\forall\, AbstractFireControlState;\ FireControlState_1 \bullet \\ \quad RetrFireControl \land \text{pre } InitAbstractFireControl \Rightarrow \text{pre } InitFireControl_1\end{array}$$

It is easily proved because the preconditions of both schemas are *true*.

The second proof obligation concerns the postcondition of both operations.

$$\begin{array}{l}\forall\, AbstractFireControlState;\ FireControlState_1;\ FireControlState_1' \bullet \\ \quad RetrFireControl \land \text{pre } InitAbstractFireControl \land InitFireControl_1 \Rightarrow \\ \qquad \exists\, AbstractFireControlState' \bullet RetrFireControl' \land InitAbstractFireControl\end{array}$$

This proof obligation can also be easily discarded using the one-point rule. When this rule is applied, we remove the universal quantifier, and then we are left with an implication in which the consequent is present in the antecedent. ❑

There is no special rule to handle initialisation operations. This is because the behaviour of a process is defined by its main action; there is no implicit initialisation. An initialisation schema is just a simplified way of specifying an operation like any other.

All other schema expressions are refined in pretty much the same way. Their definitions are very similar to the corresponding abstract operations except that the value assigned to $mode_1$ is also assigned to the new state component $mode_A$.

For the remaining actions, we rely on distribution of simulation. The new actions have the same structure as the original ones, but use the new schemas. By way of illustration, we present the action $FireSysStart_1$ that is simulated by $AbstractFireSysStart$.

$$\begin{array}{l}FireSysStart_1 \mathrel{\widehat=} systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\ \qquad\qquad\qquad switchLamp[LampId].systemOnLamp!on \rightarrow \\ \qquad\qquad\qquad\quad InitFireControl_1;\ FireSys_1\end{array}$$

To establish the simulation, we need Laws C.7 and C.11. Since all the output and input values, and guards are not changed, only their second proviso must be proved. They follow from Lemma 5.1 and $FireSys \preceq FireSys_1$.

$FireSysStart_1$ is the main action of $FireControl_1$, and we have just proved that it is simulated by the main action of $AbstractFireControl$.

$\bullet$ $FireSysStart_1$ **end**

This concludes this data refinement step.

Action Refinement In this step we change $FireControl_1$ so that its state is composed of two partitions: one that models the internal system and another that models the areas. We also change the actions so that the state partitions are handled separately.

process $ConcreteFireControl \mathrel{\widehat=}$ **begin**

The internal system state is composed only by its mode. The remaining components are declared as components of the areas partition of the state.

$$InternalSystemState \mathrel{\widehat=} [\, mode_1 : Mode \,]$$

$$\begin{array}{|l}
\hline
\textit{AreasState} \\
mode_A : Mode \\
controlledZones_1, activeZones_1 : AreaId \rightarrow \mathbb{P}\ ZoneId \\
discharge_1, active_1 : \mathbb{P}\ AreaId \\
\hline
\forall\, a : AreaId \bullet \\
\quad (mode_A = automatic) \Rightarrow a \in active_1 \Leftrightarrow \# activeZones_1\ a \geq 2 \\
\quad \wedge\ (mode_A = manual) \Rightarrow a \in active_1 \Leftrightarrow \# activeZones_1\ a \geq 1 \\
\quad \wedge\ activeZones_1\ a \subseteq controlledZones_1\ a \\
\quad \wedge\ controlledZones_1\ a = getZones\ a \\
\hline
\end{array}$$

The state of $FireControlState_1$ is declared as the conjunction of the two previously defined schemas.

$$\textbf{state}\ FireControlState_1 \mathrel{\widehat=} InternalSystemState \wedge AreasState$$

The first group of paragraphs access only $mode_1$. It is initialised to *automatic*.

$$InitInternalSystem \mathrel{\widehat=} [\, InternalSystemState' \mid mode_1' = automatic \,]$$

The schema *SwitchInternalSystemMode* receives the new mode as argument, and switches the *InternalSystem* mode.

$$\begin{array}{|l}
\hline
\textit{SwitchInternalSystemMode} \\
\Delta InternalSystemState \\
\Xi AreasState \\
nm? : Mode \\
\hline
mode_1' = nm? \\
\hline
\end{array}$$

Similarly, *SwitchInternalSystem2Auto* and *SwitchInternalSystem2Dis* set the mode of the *InternalSystem* to *automatic* and *disabled*, respectively.

The behaviour of this internal system is very similar to that of the abstract one (Table 5.2); however, after being switched *on*, it initialises only $mode_1$ and behaves like action $FireSys_2$. All the operations related to the areas are no longer controlled by the internal system actions, but by the areas actions. Here, they are handled by a different set of actions, that we present below.

For instance, consider the action $FireSysStart_2$ below.

$$\begin{array}{l}
FireSysStart_2 \mathrel{\widehat=} systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\qquad\qquad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\qquad\qquad\quad InitInternalSystem;\ FireSys_2
\end{array}$$

When a synchronisation on *modeSwitch* happens, only the *InternalSystem* mode is switched by action $FireSys_2$. Furthermore, since the informations about the areas are no longer in this partition, following a *detection* communication, this action does not activate the area in which the detection occurred. If the actuators are replaced, this action switches the corresponding lamp *on*, switches only $mode_1$ to *automatic*, and waits to be *reset*. The behaviour, if any *fault* happens, is not changed.

$$
\begin{array}{rl}
FireSys_2 \mathrel{\widehat=} & systemState!fireSys_s \rightarrow \\
& \quad modeSwitch?nm \rightarrow SwitchInternalSystemMode;\ FireSys_2 \\
& \quad \Box\ detection?nz \rightarrow switchLamp[ZoneId].nz!on \rightarrow \\
& \qquad alarm!firstStage \rightarrow \left(\begin{array}{l} (mode_1 = manual)\ \&\ Manual_2 \\ \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\
& \quad \Box\ actuatorsReplaced \rightarrow switchLamp[LampId].circFaultLamp!off \rightarrow \\
& \qquad SwitchInternalSystem2Auto;\ Reset_2 \\
& \quad \Box\ fault?faultId \rightarrow switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
& \qquad switchBuzzer!on \rightarrow FireSys_2
\end{array}
$$

The second group of paragraphs is concerned with the areas. They are initialised in *automatic* mode; furthermore, there are no active zones, no *discharge* has occurred, and no area is *active*.

$$
\begin{array}{|l}
\hline
InitAreas \\
AreasState' \\
\hline
mode'_A = automatic \land discharge'_1 = \varnothing \\
activeZones'_1 = \{a : AreaId \bullet a \mapsto \varnothing\} \\
\hline
\end{array}
$$

The areas mode can be switched to a given mode with schema *SwitchAreasMode*. The areas mode can also be switched to *automatic* or *disabled* using *SwitchAreas2AutomaticMode* and *SwitchAreas2DisabledMode*, respectively.

$$
\begin{array}{|l}
\hline
SwitchAreasMode \\
\Delta AreasState;\ \Xi InternalSystemState;\ nm? : Mode \\
\hline
mode'_A = nm? \land activeZones'_1 = activeZones_1 \\
discharge'_1 = discharge_1 \\
\hline
\end{array}
$$

The schema *ActivateZoneAS* includes a given zone $nz?$ in the set of active zones of the area that controls $nz?$.

$$
\begin{array}{|l}
\hline
ActivateZoneAS \\
\Delta AreasState;\ \Xi InternalSystemState;\ nz? : ZoneId \\
\hline
mode'_A = mode_A \land discharge'_1 = discharge_1 \\
activeZones'_1 = activeZones_1 \oplus \{a : AreaId \\
\qquad\qquad\qquad\qquad \mid nz? \in controlledZones_1\ a \\
\qquad\qquad\qquad\qquad \bullet a \mapsto activeZones_1\ a \cup \{nz?\}\} \\
\hline
\end{array}
$$

Initially, the areas synchronise on *switchOn*, initialise the state, and start their cycle.

$$StartAreas \mathrel{\widehat=} switch \rightarrow InitAreas;\ AreasCycle$$

In *AreasCycle*, the actuators can be replaced, setting the mode to *automatic*, and the areas wait to be *reset*. If the system mode is switched, so is the areas mode. Any detection in a zone *nz* leads to the activation of *nz*; the behaviour afterwards depends on the *Areas* mode.

$$\begin{aligned} AreasCycle \mathrel{\widehat=}\ & actuatorsReplaced \rightarrow SwitchAreas2AutomaticMode; ResetAreas \\ & \Box\ modeSwitch?nm \rightarrow SwitchAreasMode; AreasCycle \\ & \Box\ detection?nz \rightarrow ActivateZoneAS; \\ & \qquad\qquad\qquad\qquad (mode_A = automatic)\ \&\ AutoAreas \\ & \qquad\qquad\qquad\qquad \Box\ (mode_A = manual)\ \&\ ManualAreas \end{aligned}$$

...

As for the paragraphs of the internal system, the areas have an action corresponding to each action in the abstract system (Table 5.2); the remaining actions are omitted here.

The main action of process *ConcreteFireControl* is the parallel composition of the actions $FireSysStart_2$ and *StartAreas*. These actions actually represent the initial actions of each partition within the process. They synchronise on the channels in the set Σ_2. All the synchronisation events between the internal system and the areas are hidden in the main action.

- $(FireSysStart_2 \llbracket \alpha(InternalSystemState) \mid \Sigma_2 \mid \alpha(AreasState) \rrbracket StartAreas)$
  $\setminus GasDischargeSync$ **end**

Action $FireSysStart_2$ may modify only the components of *InternalSystemState*, and action *StartAreas* may modify only the components of *AreasState*.

Despite the fact that this is a significant refinement step, it involves no change of data representation. In order to prove that this is a valid refinement, we must prove that the main action of process *ConcreteFireControl* refines the main action of process $FireControl_1$; however, they are defined using mutual recursion, and for this reason, we use the result below in the proof. The symbol $\sqsubseteq_{\mathcal{V}}$ represents the vectorial refinement, which is defined as the individual action refinement of the corresponding actions in each vector.

Definition 5.1 (Vectorial Refinement) *For two vector of actions* $V_1 = [a_1, \ldots, a_n]$ *and* $V_2 = [c_1, \ldots, c_n]$, $V_1 \sqsubseteq_{\mathcal{V}} V_2$ *if, and only if,* $a_i \sqsubseteq_{\mathcal{A}} c_i$ *for all* i *in* $1 \mathinner{..} n$.

In order to prove that a vector of actions S_S as defined below is refined by a vector of actions $[Y_0, \ldots, Y_n]$, it is enough to show that, for each action N_i in S_S, we can prove that its definition F_i, if we replace $N_0, \ldots, N_n$ with $Y_0, \ldots, Y_n$ in F_i, is refined by Y_i.

Theorem 5.1 (Refinement of Mutually Recursive Actions) *For a given vector of actions* S_S *defined in the form* $S_S \mathrel{\widehat=} [N_0, \ldots, N_n]$, *where* $N_i \mathrel{\widehat=} F_i(N_0, \ldots, N_n)$:

$$S_S \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n] \Leftarrow \left(\begin{array}{l} F_0[Y_0, \ldots, Y_n / N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_0, \\ \ldots, \\ F_n[Y_0, \ldots, Y_n / N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_n \end{array} \right)$$

This result is proved in Appendix D.1.

We want to prove the following proposition.

$$\mathit{FireSysStart}_1 \sqsubseteq_{\mathcal{A}} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GasDischargeSync}$$

Here, $\parallel$ stands for $\llbracket \alpha(\mathit{InternalSystemState}) \mid \Sigma_2 \mid \alpha(\mathit{AreasState}) \rrbracket$.

As $\mathit{FireSysStart}_1$ is defined using mutual recursion, we use the Theorem D.1, with S_S as the following vector including all actions involved in the definition of $\mathit{FireSysStart}_1$, to prove this refinement.

$$S_S = [\mathit{FireSysStart}_1, \mathit{FireSys}_1, \ldots]$$

The vector $[Y_0, \ldots, Y_n]$ includes the parallel composition below.

$$(\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GasDischargeSync}$$

Furthermore, it also contains all the refinements of each action in S_S as a parallel composition of the same form: with the same partition, the same synchronisation set, and the same hiding.

To prove this refinement, however, using Theorem D.1, we need a modified S_S, in which some actions are preceded by an assumption. We introduce these assumptions using Law C.40.

$$\begin{array}{l} [\mathit{FireSysStart}_1, \mathit{FireSys}_1, \ldots] \\ \sqsubseteq_{\mathcal{A}} [C.40] \\ [\mathit{FireSysStart}_1, \{\mathit{mode}_1 = \mathit{mode}_A\};\ \mathit{FireSys}_1, \ldots] \end{array}$$

Although long, the proof obligation raised by this law application is trivial; we omit it here, for the sake of conciseness.

By way of illustration, let us consider the action $\mathit{FireSys}_1$: it is invoked by actions $\mathit{FireSysStart}$, Auto_1, Reset_1, $\mathit{FireSysD}_1$, and recursively by itself. However, before most of these invocations, we have either as state initialisation, or $\mathit{SwitchFireControlMode}_1$. The assumptions presented below may be introduced using laws C.28 and C.29.

$$\begin{array}{l} \{\mathit{mode}_1 = \mathit{automatic} \land \mathit{mode}_A = \mathit{automatic}\} \\ \{\mathit{mode}_1 = \mathit{newMode}? \land \mathit{mode}_A = \mathit{newMode}?\} \end{array}$$

Finally, using law C.36, we may replace these two assumptions by $\{\mathit{mode}_1 = \mathit{mode}_A\}$, since both predicates imply this equality. The only point in which no operation is present before the invocation is in the action $\mathit{FireSys}_1$ itself. However, no operation is invoked before this recursive invocation, and hence, the state does not change.

The remaining assumption introductions can be proved in a very similar way. We have that the components mode_1 and mode_A are always changed together and to the same values. This allows us to introduce the assumption $\{\mathit{mode}_1 = \mathit{mode}_A\}$ as explained above. Then, we may distribute this assumption using the distribution laws presented in Appendix C.

Using Theorem D.1 we get the following result.

$$\left(\begin{array}{l}\left[\begin{array}{l}\mathit{FireSysStart}_1,\\ \{\mathit{mode}_1 = \mathit{mode}_A\};\ \mathit{FireSys}_1,\\ \ldots\end{array}\right]\\ \sqsubseteq_{\mathcal{A}}\\ \left[\begin{array}{l}(\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GasDischargeSync},\\ (\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GasDischargeSync},\\ \ldots\end{array}\right]\end{array}\right)$$
$$\Leftarrow$$
$$\left(\begin{array}{lr}\mathit{FireSysStart}_1[\mathit{subst}] \sqsubseteq_{\mathcal{A}} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GasDischargeSync}, & (1)\\ \mathit{FireSys}_1[\mathit{subst}] \sqsubseteq_{\mathcal{A}} (\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GasDischargeSync}, \ldots & (2)\end{array}\right)$$

Here, *subst* corresponds to the following substitution.

$$\begin{array}{rl}\mathit{subst} = & [\,((\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GasDischargeSync})/\mathit{FireSysStart}_1]\\ & [\,((\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GasDischargeSync})/\mathit{FireSys}_1]\\ & \ldots\end{array}$$

Lemmas 5.2 and 5.3 prove refinements (1) and (2), respectively.

Lemma 5.2 (1)

$$\mathit{FireSysStart}_1[\mathit{subst}] \sqsubseteq_{\mathcal{A}} (\mathit{FireSysStart}_2 \parallel \mathit{StartAreas}) \setminus \mathit{GasDischargeSync}$$

Proof. We start the refinement using the definitions of $\mathit{FireSysStart}_1$ and substitution.

$$\begin{array}{l}\mathit{FireSysStart}_1[\mathit{subst}]\\ = [\text{Definition of } \mathit{FireSysStart}_1, \text{Definition of Substitution}]\\ \mathit{systemState}!\mathit{fireSysStart}_s \to \mathit{switch} \to \mathit{switchLamp}[\mathit{LampId}].\mathit{systemOnLamp}!\mathit{on} \to\\ \qquad \mathit{InitFireControl}_1;\ (\mathit{FireSys}_2 \parallel \mathit{AreasCycle}) \setminus \mathit{GasDischargeSync}\end{array}$$

First, we may expand the hiding since the channels *switchLamp*, *switchOn*, and *systemState* are not in *GasDischargeSync*.

$$\begin{array}{l}= [C.120, C.125]\\ \left(\begin{array}{l}\mathit{systemState}!\mathit{fireSysStart}_s \to \mathit{switchOn} \to\\ \qquad \mathit{switchLamp}[\mathit{LampId}].\mathit{systemOnLamp}!\mathit{on} \to\\ \qquad\qquad \mathit{InitFireControl}_1;\ (\mathit{FireSys}_2 \parallel \mathit{AreasCycle})\end{array}\right) \setminus \mathit{GasDischargeSync}\end{array}$$

The schema $\mathit{InitFireControl}_1$ can be written as the sequential composition of two other schemas as follows. In [27], a refinement law is provided to introduce a schema sequence; however, in our case, we have a initialisation schema, which has no reference to the initial state. For this reason, we use a new law that is similar to the one in [27]. Some trivial proof obligations are omitted.

$$
\begin{array}{l}
= [C.72] \\
\left(\begin{array}{l}
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad InitInternalSystem;\ InitAreas;\ (FireSys_2 \parallel AreasCycle)
\end{array}\right) \\
\setminus GasDischargeSync
\end{array}
$$

Each one of the newly inserted schema operations writes in a different partition of the parallel composition that follows them. For this reason, we may distribute them over the parallel composition. Again, two new laws are used: the first moves a (guarded) schema expression to one side of the parallel composition; commutativity of parallel composition is also provided as a new law.

$$
\begin{array}{l}
= [C.73, C.76] \\
\left(\begin{array}{l}
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad ((InitInternalSystem;\ FireSys_2) \parallel (InitAreas;\ AreasCycle))
\end{array}\right) \\
\setminus GasDischargeSync
\end{array}
$$

Next, we move the *switchLamp* event to the internal system side of the parallel composition. This step is valid because all the initial channels of *InitAreas*; *AreasCycle* are in Σ_2, and *switchLamp* is not.

$$
\begin{array}{l}
= [C.84] \\
\left(\begin{array}{l}
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad InitInternalSystem;\ FireSys_2
\end{array}\right) \\
\parallel \\
\left(\begin{array}{l}
InitAreas; \\
\quad AreasCycle
\end{array}\right)
\end{array}\right)
\end{array}\right) \\
\setminus GasDischargeSync
\end{array}
$$

Now, *switchOn* may be distributed over the parallel composition because it is in Σ_2.

$$
\begin{array}{l}
= [C.106] \\
\left(\begin{array}{l}
systemState!fireSysStart_s \rightarrow \\
\quad \left(\begin{array}{l}
switchOn \rightarrow \\
\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad InitInternalSystem;\ FireSys_2
\end{array}\right) \\
\quad \parallel \\
\quad (switchOn \rightarrow InitAreas;\ AreasCycle)
\end{array}\right) \setminus GasDischargeSync
\end{array}
$$

Since it is not in Σ_2, *systemState* may be moved to the internal system side of the parallel composition.

$$
\begin{array}{l}
= [C.100, C.84] \\
\left(\begin{array}{l}
\left(\begin{array}{l}
systemState!fireSysStart_s \rightarrow switchOn \rightarrow \\
\quad switchLamp[LampId].systemOnLamp!on \rightarrow \\
\quad\quad InitInternalSystem;\ FireSys_2
\end{array}\right) \\
\parallel \\
(switchOn \rightarrow InitAreas;\ AreasCycle)
\end{array}\right) \\
\setminus GasDischargeSync
\end{array}
$$

Finally, using the definitions of $FireSysStart_2$ and $StartAreas$ we conclude this proof.

$$
\begin{array}{l}
= [\text{Definition of } FireSysStart_2 \text{ and } StartAreas] \\
(FireSysStart_2 \parallel StartAreas) \setminus GasDischargeSync
\end{array}
\qquad \square
$$

The next lemma we present is the refinement of the action $FireSys_1$.

Lemma 5.3 (2)

$$
\begin{array}{l}
\{mode_1 = mode_A\};\ FireSys_1[subst] \\
\sqsubseteq_{\mathcal{A}} \\
(FireSys_2 \parallel AreasCycle) \setminus GasDischargeSync
\end{array}
$$

Proof. We start the proof using the definitions of $FireSys_1$ and substitution.

$$
\begin{array}{l}
\{mode_1 = mode_A\};\ FireSys_1[subst] \\
= [\text{Definition of } FireSys_1, \text{Definition of Substitution}] \\
\{mode_1 = mode_A\}; \\
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?nm \rightarrow \\
\quad\quad SwitchFireControlMode_1;\ (FireSys_2 \parallel AreasCycle) \setminus GasDischargeSync \\
\quad \Box\ detection?nz \rightarrow ActivateZone_1;\ switchLamp[ZoneId].nz!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ (Manual_2 \parallel ManualAreas) \setminus GasDischargeSync \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ (Auto_2 \parallel AutoAreas) \setminus GasDischargeSync \\
\quad \Box\ actuatorsReplaced \rightarrow switchLamp[LampId].circFaultLamp!off \rightarrow \\
\quad\quad SwitchFireControl2Auto_1;\ (Reset_2 \parallel ResetAreas) \setminus GasDischargeSync \\
\quad \Box\ fault?faultId \rightarrow switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow (FireSys_2 \parallel AreasCycle) \setminus GasDischargeSync
\end{array}
$$

Next, we expand the hiding to the whole action. This is valid because all the events involved in the expansion are not in the hidden set of channels.

$$
\begin{array}{l}
= [C.120, C.125, C.122] \\
\left(\begin{array}{ll}
\{mode_1 = mode_A\}; & \\
systemState!fireSys_s \rightarrow & \\
\quad modeSwitch?nm \rightarrow SwitchFireControlMode_1;\ (FireSys_2 \parallel AreasCycle) & (3) \\
\quad \Box\ detection?nz \rightarrow ActivateZone_1;\ switchLamp[ZoneId].nz!on \rightarrow & (4) \\
\quad\quad alarm!firstStage \rightarrow & \\
\quad\quad\quad (mode_1 = manual)\ \&\ (Manual_2 \parallel ManualAreas) & \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ (Auto_2 \parallel AutoAreas) & \\
\quad \Box\ actuatorsReplaced \rightarrow switchLamp[LampId].circFaultLamp!off \rightarrow & (5) \\
\quad\quad SwitchFireControl2Auto_1;\ (Reset_2 \parallel ResetAreas) & \\
\quad \Box\ fault?faultId \rightarrow switchLamp[LampId].(getLampId\ faultId)!on \rightarrow & (6) \\
\quad\quad switchBuzzer!on \rightarrow (FireSys_2 \parallel AreasCycle) &
\end{array}\right) \\
\setminus\ GasDischargeSync
\end{array}
$$

Next, we aim at the refinement of each branch to a parallel composition in order to be able to apply the exchange Law C.85. First, we refine (3) as follows: $SwitchFireControlMode_1$ can be written as the sequential composition of the two schemas $SwitchInternalSystemMode$ and $SwitchAreasMode$.

$$
\begin{array}{l}
(3) \sqsubseteq_{\mathcal{A}} [C.71] \\
\quad modeSwitch?nm \rightarrow SwitchInternalSystemMode;\ SwitchAreasMode; \\
\quad\quad (FireSys_2 \parallel AreasCycle)
\end{array}
$$

The two schemas can be moved to different sides of the parallel composition.

$$
\begin{array}{l}
= [C.76, C.73] \\
modeSwitch?nm \rightarrow \\
\quad ((SwitchInternalSystemMode;\ FireSys_2) \parallel (SwitchAreasMode;\ AreasCycle))
\end{array}
$$

Finally, as $modeSwitch$ is in Σ_2, we may distribute this event over the parallel composition. Here, a new law (distribution of input channels over parallel composition) is used.

$$
\begin{array}{l}
= [C.108] \\
\left(\begin{array}{l}
modeSwitch?nm \rightarrow \\
\quad SwitchInternalSystemMode; \\
\quad\quad FireSys_2
\end{array}\right) \parallel \left(\begin{array}{l}
modeSwitch?nm \rightarrow \\
\quad SwitchAreasMode;\ AreasCycle
\end{array}\right)
\end{array}
$$

For (4), we first use the assumption laws in order to move the assumption into the action.

$$
\begin{array}{l}
(4) \sqsubseteq_{\mathcal{A}} [C.45, C.37, C.35, C.132, C.47, C.53] \\
\quad detection?nz \rightarrow ActivateZone_1;\ switchLamp[ZoneId].nz!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \{mode_1 = mode_A\}; \\
\quad\quad\quad \{mode_1 = mode_A\};\ (mode_1 = manual)\ \&\ (Manual_2 \parallel ManualAreas) \\
\quad\quad\quad \Box\ \{mode_1 = mode_A\};\ (mode_1 = automatic)\ \&\ (Auto_2 \parallel AutoAreas)
\end{array}
$$

Next, we use the assumption to change the guards.

$$
\begin{array}{l}
= [C.34] \\
detection?nz \to ActivateZone_1;\ switchLamp[ZoneId].nz!on \to \\
\quad alarm!firstStage \to \\
\quad\quad \{mode_1 = mode_A\}; \\
\quad\quad\quad \{mode_1 = mode_A\}; \\
\quad\quad\quad\quad (mode_1 = manual \land mode_A = manual)\ \& \\
\quad\quad\quad\quad\quad (Manual_2 \parallel ManualAreas) \\
\quad\quad\quad \Box\ \{mode_1 = mode_A\}; \\
\quad\quad\quad\quad (mode_1 = automatic \land mode_A = automatic)\ \& \\
\quad\quad\quad\quad\quad (Auto_2 \parallel AutoAreas)
\end{array}
$$

The assumptions can then be absorbed by the guards.

$$
\begin{array}{l}
= [C.30, C.57, C.35, C.132] \\
detection?nz \to ActivateZone_1;\ switchLamp[ZoneId].nz!on \to \\
\quad alarm!firstStage \to \\
\quad\quad \{mode_1 = mode_A\}; \\
\quad\quad\quad (mode_1 = mode_A \land mode_1 = manual \land mode_A = manual)\ \& \\
\quad\quad\quad\quad (Manual_2 \parallel ManualAreas) \\
\quad\quad\quad \Box\ (mode_1 = mode_A \land mode_1 = automatic \land mode_A = automatic)\ \& \\
\quad\quad\quad\quad (Auto_2 \parallel AutoAreas)
\end{array}
$$

Now, we distribute the guards over the parallel composition.

$$
\begin{array}{l}
= [C.64] \\
detection?nz \to ActivateZone_1;\ switchLamp[ZoneId].nz!on \to \\
\quad alarm!firstStage \to \{mode_1 = mode_A\}; \\
\quad\quad \left(\begin{array}{l} \left(\begin{array}{l} mode_1 = mode_A \land \\ mode_1 = manual \end{array} \right) \& \\ \quad Manual_2 \end{array} \right) \parallel \left(\begin{array}{l} \left(\begin{array}{l} mode_1 = mode_A \land \\ mode_A = manual \end{array} \right) \& \\ \quad ManualAreas \end{array} \right) \\
\quad\quad \Box \left(\begin{array}{l} \left(\begin{array}{l} mode_1 = mode_A \land \\ mode_1 = automatic \end{array} \right) \& \\ \quad Auto_2 \end{array} \right) \parallel \left(\begin{array}{l} \left(\begin{array}{l} mode_1 = mode_A \land \\ mode_A = automatic \end{array} \right) \& \\ \quad AutoAreas \end{array} \right)
\end{array}
$$

The guards are mutually exclusive; we may apply an exchange law that simplifies them.

$$
\begin{array}{l}
= [C.85, C.37, C.34, C.35, C.132] \\
detection?nz \to ActivateZone_1;\ switchLamp[ZoneId].nz!on \to \\
\quad alarm!firstStage \to \\
\quad\quad \left(\begin{array}{l} (mode_1 = manual)\ \& \\ \quad Manual_2 \\ \Box\ (mode_1 = automatic)\ \& \\ \quad Auto_2 \end{array} \right) \parallel \left(\begin{array}{l} (mode_A = manual)\ \& \\ \quad ManualAreas \\ \Box\ (mode_A = automatic)\ \& \\ \quad AutoAreas \end{array} \right)
\end{array}
$$

Next, we move the outputs channels to the left-hand side of the parallel composition. This follows from the fact that the initial channels of both *ManualAreas* and *AutoAreas* are in Σ_2, and *alarm* and *switchLamp* are not.

$$
\begin{array}{l}
= [C.100, C.84] \\
detection?nz \rightarrow ActivateZone_1; \\
\quad \left(\begin{array}{l}
switchLamp[ZoneId].nz!on \rightarrow \\
\quad alarm!firstStage \rightarrow \\
\quad\quad (mode_1 = manual)\ \& \\
\quad\quad\quad Manual_2 \\
\quad\quad \Box\ (mode_1 = automatic)\ \& \\
\quad\quad\quad Auto_2
\end{array}\right) \\
\quad \| \\
\quad \left(\begin{array}{l}
(mode_A = manual)\ \& \\
\quad ManualAreas \\
\Box\ (mode_A = automatic)\ \& \\
\quad AutoAreas
\end{array}\right)
\end{array}
$$

The schema $ActivateZone_1$ can easily be transformed to $ActivateZoneAS$ using the schema calculus. The resulting schema can also be distributed over the parallel composition. Finally, channel *detection* can be distributed over the parallel composition, since it is in Σ_2.

$$
\begin{array}{l}
= [\text{Schema Calculus}, C.76, C.73, C.108] \\
\left(\begin{array}{l}
detection?nz \rightarrow switchLamp[ZoneId].nz!on \rightarrow alarm!firstStage \rightarrow \\
\quad (mode_1 = manual)\ \& \\
\quad\quad Manual_2 \\
\quad \Box\ (mode_1 = automatic)\ \& \\
\quad\quad Auto_2
\end{array}\right) \\
\| \\
\left(\begin{array}{l}
detection?nz \rightarrow ActivateZoneAS; \\
\quad \left(\begin{array}{l}
(mode_A = manual)\ \&\ ManualAreas \\
\Box\ (mode_A = automatic)\ \&\ AutoAreas
\end{array}\right)
\end{array}\right)
\end{array}
$$

Using similar strategies, we refine (5) and (6) to the following external choice.

$$
\begin{array}{l}
(5,6) = [\ldots] \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
actuatorsReplaced \rightarrow \\
\quad switchLamp[LampId].circFaultLamp!off \rightarrow \\
\quad\quad SwitchInternalSystem2Auto;\ Reset_2
\end{array}\right) \\
\| \\
(actuatorsReplaced \rightarrow SwitchAreas2AutomaticMode;\ ResetAreas)
\end{array}\right) \\
\quad \Box \left(\begin{array}{l}
fault?faultId \rightarrow \\
\quad switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \\
\quad\quad \| \\
\quad\quad AreasCycle
\end{array}
$$

We are left with the external choice of parallel actions. Since the initial channels of the first three parallel actions are in the set Σ_2, we may apply the exchange law as follows.

$$
\begin{array}{l}
= [C.85] \\
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
modeSwitch?nm \rightarrow SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ detection?nz \rightarrow switchLamp[ZoneId].nz!on \rightarrow \\
\qquad alarm!firstStage \rightarrow \left(\begin{array}{l} (mode_1 = manual)\ \&\ Manual_2 \\ \Box\ (mode_1 = automatic)\ \&\ Auto_2 \end{array} \right) \\
\Box\ actuatorsReplaced \rightarrow switchLamp[LampId].circFaultLamp!off \rightarrow \\
\qquad SwitchInternalSystem2Auto;\ Reset_2
\end{array} \right) \\
\parallel \\
\left(\begin{array}{l}
modeSwitch?nm \rightarrow SwitchAreasMode;\ AreasCycle \\
\Box\ detection?nz \rightarrow ActivateZoneAS; \\
\qquad (mode_A = manual)\ \&\ ManualAreas \\
\qquad \Box\ (mode_A = automatic)\ \&\ AutoAreas \\
\Box\ actuatorsReplaced \rightarrow SwitchAreas2AutomaticMode;\ ResetAreas
\end{array} \right)
\end{array} \right) \\
\quad \Box \left(\begin{array}{l}
fault?faultId \rightarrow \\
\qquad switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow FireSys_2
\end{array} \right) \parallel AreasCycle
\end{array}
$$

Using the associativity of external choice, we have that the right-hand side of the first parallel composition corresponds to the definition of the action *AreasCycle*. So, we have that both branches of the external choice have this action as the right-hand side of the parallel composition. Since all the initials of *AreasCycle* are in Σ_2, we may apply the distribution of parallel composition over external choice.

$$
\begin{array}{l}
= [C.87] \\
systemState!fireSys_s \rightarrow \\
\quad \left(\begin{array}{l}
modeSwitch?nm \rightarrow SwitchInternalSystemMode;\ FireSys_2 \\
\Box\ detection?nz \rightarrow switchLamp[ZoneId].nz!on \rightarrow \\
\qquad alarm!firstStage \rightarrow \\
\qquad\qquad (mode_1 = manual)\ \&\ Manual_2 \\
\qquad\qquad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\Box\ actuatorsReplaced \rightarrow \\
\qquad switchLamp[LampId].circFaultLamp!off \rightarrow \\
\qquad\qquad SwitchInternalSystem2Auto;\ Reset_2 \\
\Box\ fault?faultId \rightarrow \\
\qquad switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\qquad\qquad switchBuzzer!on \rightarrow FireSys_2
\end{array} \right) \\
\quad \parallel \\
\quad AreasCycle
\end{array}
$$

Finally, we can distribute the communication that uses *systemState* and use the definition of $FireSys_2$ to conclude our proof. Again, this is valid because all the initials of *AreasCycle* are in Σ_2, and *systemState* is not.

$$
\begin{array}{l}
= [C.100, C.84] \\
\left(\begin{array}{l}
systemState!fireSys_s \rightarrow \\
\quad modeSwitch?nm \rightarrow SwitchInternalSystemMode;\ FireSys_2 \\
\quad \Box\ detection?nz \rightarrow switchLamp[ZoneId].nz!on \rightarrow \\
\quad\quad alarm!firstStage \rightarrow \\
\quad\quad\quad (mode_1 = manual)\ \&\ Manual_2 \\
\quad\quad\quad \Box\ (mode_1 = automatic)\ \&\ Auto_2 \\
\quad \Box\ actuatorsReplaced \rightarrow \\
\quad\quad switchLamp[LampId].circFaultLamp!off \rightarrow \\
\quad\quad\quad SwitchInternalSystem2Auto;\ Reset_2 \\
\quad \Box\ fault?faultId \rightarrow \\
\quad\quad switchLamp[LampId].(getLampId\ faultId)!on \rightarrow \\
\quad\quad\quad switchBuzzer!on \rightarrow FireSys_2
\end{array}\right) \parallel AreasCycle \\
= [\text{Definition of } FireSys_2] \\
(FireSys_2 \parallel AreasCycle) \setminus GasDischargeSync \qquad \Box
\end{array}
$$

Using these lemmas, and those related to the remaining actions, which are omitted here, we prove that $FireControl_1$ is refined by *ConcreteFireControl*.

Process Refinement We partitioned the state of the process $FireControl_1$ into the schemas *InternalSystemState* and *AreasState*. Each partition has its own set of paragraphs, which are disjoint, since, no action in one changes a state component in the other. Furthermore, the main action of the refined process is defined in terms of these two partitions. Therefore, we may apply Law C.146 in order to split process *ConcreteFireControl* into two independent processes as follows.

$$
\begin{array}{l}
\textbf{process}\ ConcreteFireControl \mathrel{\widehat=} \\
\quad (InternalSystem \mathbin{[\![} \Sigma_2 \mathbin{]\!]} Areas) \setminus GasDischargeSync
\end{array}
$$

The *ConcreteFireControl* is redefined as the parallel composition of *InternalSystem* and *Areas*. Their definitions can be deduced from the definition of *ConcreteFireControl*.

5.3.3 Second Iteration: splitting *InternalSystem* into two controllers

In this iteration, we split *InternalSystem* into two separated partitions: the first one corresponds to the *FireControl* controller, and the other the *DisplayController* (see Figure 5.5). The fire control internal system state is left unchanged, and so this iteration does not require a data refinement.

Action Refinement We rewrite the actions so that the *FireControl* paragraphs no longer deal with the display events, which are dealt by *DisplayController*. The display controller has no state at all, so the new state is defined as follows.

$$\begin{array}{l} \textbf{process}\ \mathit{ConcreteInternalSystem} \mathrel{\widehat=} \textbf{begin} \\ \quad \mathit{FireControlState} \mathrel{\widehat=} [\, \mathit{mode}_1 : \mathit{Mode} \,] \\ \quad \textbf{state}\ \mathit{InternalSystemState}_1 \mathrel{\widehat=} \mathit{FireControlState} \end{array}$$

The operations over the *InternalSystemState* are slightly changed: they are renamed and affect the *FireControlState*, which is the same as the *InternalSystemState*. Their definitions, and those of all actions over *FireControlState* have the same definition and description as those of the process *FireControl* in the target design. Also, the display paragraphs are those of *DisplayController*, which can be found in Section 5.3.1.

The main action of the *ConcreteInternalSystem* is as follows.

$$\begin{array}{l} \bullet \left(\begin{array}{l} \mathit{FireSysStart} \\ [\![\alpha(\mathit{FireControlState}) \mid \Sigma_2 \mid \alpha(\mathit{DisplayControllerState})]\!] \\ \mathit{StartDisplay} \end{array} \right) \setminus \mathit{DisplaySync} \\ \textbf{end} \end{array}$$

We have the parallel composition of action *FireSysStart* and *StartDisplay*, with the channels used exclusively for their communication hidden. Again, since $\mathit{FireSysStart}_2$, *FireSysStart*, and *StartDisplay* are defined using mutual recursion, we use Theorem D.1 to prove that the process *InternalSystem* is refined by *ConcreteInternalSystem*. The details can be found in [71].

Process Refinement Each partition in *ConcreteInternalSystem* has its own set of paragraphs, which are disjoint. Furthermore, we define the main action of the refined process in terms of these two partitions. Applying Law C.146, we get the following result.

$$\begin{array}{l} \textbf{process}\ \mathit{ConcreteInternalSystem} \mathrel{\widehat=} \\ \quad (\mathit{FireControl} \,[\![\, \Sigma_1 \,]\!]\, \mathit{DisplayController}) \setminus \mathit{DisplaySync} \end{array}$$

The processes *FireControl* and the *DisplayController* were already described in the specification of the concrete system in Section 5.3.1.

5.3.4 Third Iteration: splitting the *Areas* into individual *Area*s

This last iteration aims at splitting *Areas* in individual processes *Area* for each area.

Data Refinement First, we must apply a data refinement to the original process *Areas*.

$$\textbf{process}\ \mathit{Areas}_1 \mathrel{\widehat=} \textbf{begin}$$

We introduce a local state *AreaState* of an individual *Area*. Its definition is very similar to that of the concrete system, but includes an identifier $id : \mathit{AreaId}$. The new global state $\mathit{AreasState}_1$ is defined as a total function from *AreaId* to local states. The invariant handles the new data structure.

state

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AreasState}_1} \\
\hline
areas : AreaId \rightarrow AreaState \\
\hline
\forall a : AreaId \bullet \\
\quad (areas\ a).id = a \\
\quad \wedge ((areas\ a).mode = automatic) \Rightarrow \\
\qquad (areas\ a).active = true \Leftrightarrow \#(areas\ a).activeZones \geq 2 \\
\quad \wedge ((areas\ a).mode = manual) \Rightarrow \\
\qquad (areas\ a).active = true \Leftrightarrow \#(areas\ a).activeZones \geq 1 \\
\quad \wedge (areas\ a).activeZones \subseteq (areas\ a).controlledZones \\
\quad \wedge (areas\ a).controlledZones = getZones\ a \\
\hline
\end{array}
$$

The retrieve relation is very simple and is defined below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RetrieveAreas}} \\
\hline
AreasState;\ AreasState_1 \\
\hline
\forall a : AreaId \bullet (areas\ a).mode = mode_A \\
\qquad \wedge (areas\ a).controlledZones = controlledZones_1\ a \\
\qquad \wedge (areas\ a).activeZones = activeZones_1\ a \\
\qquad \wedge (areas\ a).discharge = true \Leftrightarrow a \in discharge_1 \\
\qquad \wedge (areas\ a).active = true \Leftrightarrow a \in active_1 \\
\hline
\end{array}
$$

The mode in each of the local areas is that of *Areas*; the controlled and active zones of an area is defined as the corresponding image in the global state; a discharge has occurred in an area, if it is in $discharge_1$; and finally, the area is active if it is in $active_1$.

We introduce the paragraphs related to the local state *AreaState*. Basically, we have a corresponding local action for each global action. They are identical to those presented within the process *Area* in the concrete system, and are omitted at this point for conciseness.

Next, we redefine each of the global operations. Basically, all global operations have an effect in each of the individual local states. For instance, *InitAreas* is refined below.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitAreas}_1} \\
\hline
AreasState_1' \\
\hline
\forall a : AreaId \bullet (areas'\ a).activeZones = \varnothing \\
\qquad \wedge (areas'\ a).discharge = false \\
\qquad \wedge (areas'\ a).mode = automatic \\
\hline
\end{array}
$$

The proof of the simulations are simple, but long. As before, for the main action, we rely on the fact that forwards simulation distributes through action constructors. The new actions have the same structure as the original ones, but use new schema actions.

$$
\begin{array}{l}
StartAreas_1 \mathrel{\widehat=} switchOn \rightarrow InitAreas_1;\ AreasCycle_1 \\
AreasCycle_1 \mathrel{\widehat=} \ actuatorsReplaced \rightarrow SwitchAreas2AutomaticMode_1; ResetAreas_1 \\
\qquad\qquad\quad \Box\ modeSwitch?nm \rightarrow SwitchAreasMode_1;\ AreasCycle_1 \\
\qquad\qquad\quad \Box\ detection?nz \rightarrow ActivateZoneAS_1; \\
\qquad\qquad\qquad (\forall a : AreaId \bullet (areas\ a).mode = automatic)\ \&\ AutoAreas_1 \\
\qquad\qquad\qquad \Box\ (\forall a : AreaId \bullet (areas\ a).mode = manual)\ \&\ ManualAreas_1
\end{array}
$$

Since all the output and input values are not changed, in the application of Law C.7 we only rely on distribution. On the other hand, all the guards are changed. Both provisos raised by Law C.11 need to be proved. For instance, to prove the refinement of simulation for $AreasCycle_1$ we need the following lemma.

Lemma 5.4 *For any mode M,*

$$
\begin{array}{l}
\forall AreasState;\ AreasState_1 \bullet RetrieveAreas \Rightarrow \\
\quad mode_A = M \Leftrightarrow \forall a : AreaId \bullet (areas\ a).mode = M
\end{array}
$$

Proof. The proof of this lemma follows from predicate calculus, using the retrieve relation $RetrieveAreas$ to relate $mode_A$ with each individual area's $mode$. ❑

The main action of the areas, $Areas_1$, is the simulation of the original action.

$\bullet\ StartAreas_1$
end

This concludes this data refinement step.

Action Refinement In order to apply a process refinement that splits the $Areas_1$ process into individual areas, we redefine each of the paragraphs within $Areas_1$ as a promotion of the corresponding original one.

The local paragraphs and the global state remain unchanged. However, a promotion schema is introduced; it relates the local state to the global one.

$$
\begin{array}{|l}
\multicolumn{1}{l}{Promotion} \\ \hline
\Delta AreasState_1;\ \Delta AreaState;\ id? : AreaId \\ \hline
\theta AreaState = areas\ id? \land areas' = areas \oplus \{id? \mapsto \theta AreaState'\} \\ \hline
\end{array}
$$

The global operations are refined to a definition in terms of the corresponding local operations. For instance, the initialisation is refined as follows.

$$InitAreas_1 \mathrel{\widehat=} \forall id? : AreaId \bullet InitArea \land Promotion$$

This can be proved using the action refinement laws presented in Appendix C. The redefinition of the remaining operations are trivially similar and omitted here.

Each action is defined as an iterated parallel composition of the promotion of the corresponding local operation, but substituting the area *id* by the indexing variable i. Each branch of the parallel composition may change its corresponding local state $areas\ i$; the remaining branches j, such that $j \neq i$, may change the remaining local states $areas\ j$. For instance, the actions $StartAreas_1$ and $AreasCycle_1$ can be rewritten as follows.

$$
\begin{array}{l}
StartAreas_2 \mathrel{\widehat=} \\
\quad \llbracket \Sigma_{areas} \rrbracket\ i : AreaId \bullet \llbracket \alpha\,(areas\ i) \rrbracket \bullet (\mathbf{promote_2}\ StartArea)\,[id, id? := i, i]
\end{array}
$$

The remaining actions are rewritten in a very similar way. Finally, we replace the main action.

$\bullet\ StartAreas_2$
end

Since $StartAreas_1$ and $StartAreas_2$ use mutual recursion, we use Theorem D.1 again to prove that $Areas_2$ is a refinement of $Areas_1$.

Process Refinement This last process split needs the process refinement law C.147. This law applies to processes whose definition contains a local state and local operations, and a global state and global operations expressed in terms of the promotion of local paragraphs to the global state using iterated parallel operator. The application of this law creates a local process parametrised by an identifier and a global process defined as an iterated parallel composition of local processes.

We apply this law to $Areas_1$ in order to express the *Areas* process as the following parallel composition of individual *Area* processes.

$$\mathbf{process}\ ConcreteAreas \mathrel{\widehat=} \llbracket \Sigma_{areas} \rrbracket\ id : AreaId \bullet Area(id)$$

The *Area* definition corresponds to that in the concrete system.

5.4 Final Considerations

In this Chapter, we presented the development of a case study on the *Circus* refinement calculus. Using *Circus*, we were able to specify elegantly both behavioural and data aspects of an industrial scale application. With that, we demonstrate that the refinement strategy presented in [27] is also applicable to large systems. The development consisted of three iterations: the first one splits the system into a system controller and the sensors. In the second iteration, the control is subdivided into two different controllers: one for the system and one for the display. Finally, the third iteration splits the sensors into individual processes, one for each area.

The set of laws presented in [27] was not sufficient. Our case study has motivated the proposal of new refinement laws. For instance, we require some laws for inserting and distributing assumptions, and a new process refinement law. In total, more than one-hundred new laws have been identified during the development of our case study; they can be found in Appendix C. Furthermore, some laws presented in [27] were found to be incorrect, and they are corrected in this thesis (see Chapter 4). Next, the refinement of mutually recursive actions was considered, and we presented a notation used to prove refinement of such systems that results in more concise and modular proofs.

Other case studies on refinement in *Circus* have already been presented elsewhere. Woodcock and Cavalcanti present the development of a steam boiler in [104]; Freitas presents the refinement of a connection pool in [45]. However, so far, only small examples have taken a calculational approach [27, 101]. As far as we know, the case study presented in this chapter is the largest case study on the *Circus* refinement calculus.

The development presented in this chapter and all the proofs that were needed have been done by hand. In the future, we intend to use the mechanisation of our refinement laws, in order to mechanically verify our refinement.

The result of the refinement presented here does not involve only executable constructs; additional simple schema refinements using [29] were omitted here. The implementation of this case study in Java and the strategy that we devised to obtain this implementation are the subject of the next chapter.

Chapter 6

Translation to Java with Processes

In this chapter we present a strategy for implementing *Circus* programs in JCSP [99, 98]. The strategy is based on a number of translation laws, which if applied exhaustively, transform a *Circus* program into a Java program that uses the JCSP library. We assume that, before applying the translation strategy presented in this chapter, the specification of the system we want to implement has been already refined, using the *Circus* refinement strategy (Chapter 4), to meet the translation strategy's requirements discussed in Section 6.2.

First, Section 6.1 presents JCSP and some examples. Section 6.2 presents the strategy to implement *Circus* programs using JCSP: the basics of our strategy are presented from Section 6.2.1 to 6.2.6. In Section 6.2.7 we extend the types of communication considered; we deal with communication events of the form N.Expression as opposed to inputs and outputs. The translation strategy for the *Circus* indexed operator is presented in Section 6.2.8. Generic channels are considered in Section 6.2.9 and multi-synchronisation in Section 6.2.10. Finally, in Section 6.3 we discuss the translation of our case study presented in Chapter 5. Part of the material in this chapter was published in [73, 76].

6.1 JCSP

Since the facilities for concurrency in Java do not directly correspond with the idea of processes in CSP and *Circus*, we use JCSP, a library that provides a model for processes and channels. This allows us to abstract from basic monitor constructs provided by Java. In JCSP, a process is a class that implements the following Java interface.

```
interface CSProcess{ public void run(); }
```

The method `run` encodes its behaviour. We present an `Example` process below.

```
import jcsp.lang.*; // further imports
class Example implements CSProcess {
    // state information, constructors, and auxiliary methods
    public void run { /* execution of the process */ } }
```

After importing the basic JCSP classes and any other relevant classes, we declare `Example`, which may have private attributes, constructors, and auxiliary methods. We must also give the implementation of the method `run`.

Some JCSP interfaces represent channels: `ChannelInput` is the type of channels used to read objects; `ChannelOutput` is for channels used to write objects; and `AltingChannel` is for channels used in choices. Other interfaces are available, but these are the only ones used in our work.

The class `One2OneChannel`, which represents a point-to-point channel, is the simplest implementation of a channel interface provided by JCSP; multiple readers and writers are not allowed. On the other hand, `Any2OneChannel` channels allow many writers to communicate with one reader. For any type of channel, a communication happens between one writer and one reader only.

Mostly, JCSP channels communicate Java objects. For instance, in order to communicate an object `o` through a channel `c`, a writer process may declare `c` as a `ChannelOutput`, and invoke `c.write(o)`; a reader process that declares `c` as a `ChannelInput` invokes `c.read()`, which returns the communicated `Object`.

The class `Alternative` implements the choice operator. Although other types of choice are available, we use a fair choice. Only `AltingChannelInput` channels may be involved in choices. The code below reads from either channel `l` or `r`.

```
AltingChannelInput[] chs = new AltingChannelInput[]{l,r};
final Alternative alt = new Alternative(chs);
chs[alt.select()].read();
```

The channels `l` and `r` are included in an array of channels `chs`, which is given to the constructor of the `Alternative`. The method `select` waits for one or more channels to become ready, makes an arbitrary choice between them, and returns an `int` that corresponds to the index of the chosen channel in `chs`. Finally, we read from the channel located at the chosen position of `chs`.

Parallel processes are implemented using the class `Parallel`. Its constructor takes an array of `CSProcess`es and returns a `CSProcess` that is the parallel composition of its process arguments. A `run` of a `Parallel` process terminates when all its component processes terminate. For instance, the code presented below executes two processes `P_1` and `P_2` in parallel.

```
(new Parallel(new CSProcess[]{P_1,P_2})).run();
```

It creates the array of processes that run in parallel, gives it to the constructor of `Parallel`, and finally, runs the parallel composition.

The CSP constructors *Skip* and *Stop* are implemented by the classes `Skip` and `Stop`. JCSP includes other facilities beyond those available in CSP; here we concentrate on those that are relevant for our work. For more details, refer to [98].

6.2 From *Circus* to JCSP

Our strategy for translating *Circus* programs considers each paragraph individually, and in sequence. In Figure 6.1, we present an overview of the translation strategy. First,

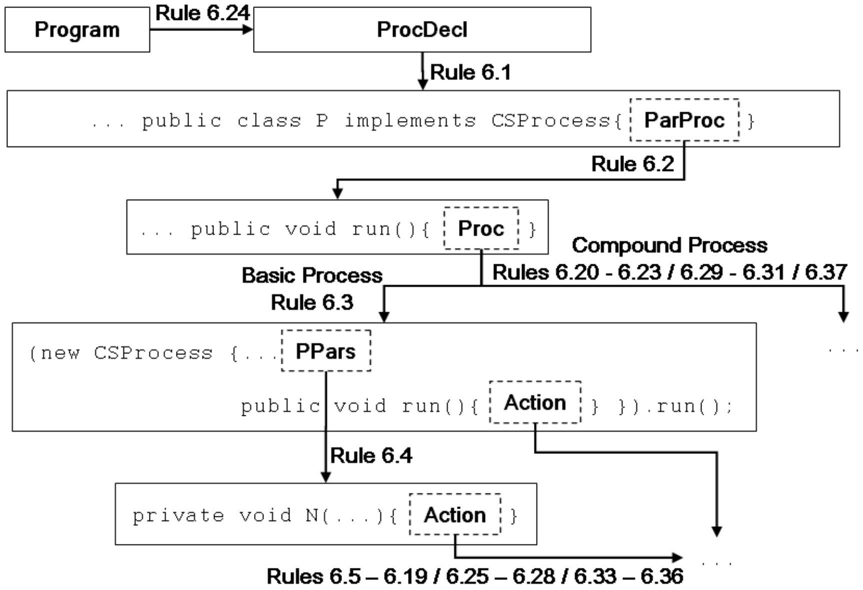


Figure 6.1: Translation Strategy Overview

for a given Program, we use a rule (6.24) that deals with the Z paragraphs and channel declarations. Each process declaration ProcDecl in the program is transformed into a new Java class (6.1). The next step (6.2) declares the class attributes, constructor, and its `run` method. Basic process definitions are translated (6.3) to the execution of a process whose private methods correspond to the translation (6.4) of actions of the original *Circus* process; the translation of the main Action, which determines the body of the method `run`, and of the Action bodies conclude the translation of basic processes (6.5-6.19, 6.25-6.28, and 6.33-6.36). Compound processes are translated using a separate set of rules (6.20-6.23, 6.29-6.31, and 6.37) that combines the translations of the basic processes.

Requirements. Only executable *Circus* programs can be translated: the technique in [27] can be used to refine specifications. Other restrictions are syntactic and can be enforced by a (mechanised) pre-processing; they are listed below.

- The *Circus* program is well-typed and well-formed.
- Paragraphs are grouped in the following order: Z paragraphs, channel declarations, and process declarations.
- Z paragraphs are axiomatic definitions of the form $v : T \mid v = e$, free types, or abbreviations.
- The only Z paragraphs inside a process declaration are axiomatic definitions of the above form.
- Variable declarations are of the form $x_1 : T_1;\ x_2 : T_2;\ \ldots;\ x_n : T_n$, and names are not reused.

- There are no nested external choices or nested guards.
- Actions in a parallel composition do not invoke any other action.
- The synchronisation sets in any parallel composition are the intersection of the sets of channels used by the parallel actions or processes.
- No channel is used by two interleaved actions or processes.
- The types used are already implemented in Java.
- There are no output guards.
- Channels involved in a multi-synchronisation are neither generic nor synchronisation channels.

Axiomatic definitions can be used to define only constants. All types, abbreviations and free types, need a corresponding Java implementation. If necessary, the *Circus* data refinement technique should be used. Nested external choices and guarded actions can be eliminated with simple refinement laws.

In a parallel composition, the actions cannot invoke any other action. This restriction can be easily satisfied by replacing any action invocation by its body with the substitution of the formal parameters for the arguments used in the invocation.

The JCSP parallel construct does not allow the definition of a synchronisation channel set. For this reason, the intersection of the alphabets determines this set: if it is not empty, we have a parallel composition; otherwise, we have actually an interleaving. JCSP does not have an interleaving construct; when possible we use the parallel construct instead.

Output guards are not implementable in JCSP. Before applying the translation strategy they must be removed applying refinement strategies like that presented in [101] for multi-synchronisation.

The output of the translation is Java code composed of several class declarations that can be split into different files and allocated in packages. For each program, we require a project name `proj`. The translation generates six packages: `proj` contains the main class, which is used to execute the system; `proj.axiomaticDefinitions` contains the class that encapsulates the translation of all axiomatic definitions; the processes are declared in the package `proj.processes`; `proj.typing` contains all the classes that implement types; and `proj.util` contains all the utility classes used by the generated code. For example, the class `RandomGenerator` is used to generate random numbers; it is used in the implementation of internal choice.

The translation uses a channel environment δ. For each channel c, it maps c to its type, or to $Sync$, if c is a synchronisation channel. We consider δ to be available throughout the translation. In order to simplify the definitions throughout this chapter, we use a non-standard representation of a channel type. For instance, the generic channel declared as **channel** $[T]c : T \times \mathbb{Z}$ is represented in this environment as the mapping $c \mapsto ([T], [T, \mathbb{Z}])$. The first list contains the typing variables and the second contains the types used in the declaration of the channel. Untyped channels are mapped to $([\,], [Sync])$.

For each process, two environments store information about channels: ν and ι for visible and hidden channels; both map channel names to an element of $ChanUse ::= I \mid O \mid A$. The constant I is used for input channels, O for output channels, and A for input channels that take part in external choices. Synchronisation channels must also be associated to one of these constants, since every JCSP channel is either an input or an output channel. If a channel c is regarded as an input channel in a process P, then it must be regarded as an output channel in any process parallel to P, and vice-versa. A multi-synchronised channel is regarded as an output channel in only one of the processes that synchronise on it; it is regarded as an input channel in the other processes.

A type environment is also considered available in the translation: the environment τ of type seq Expression lists all the types that are used in the *Circus* program which is being translated. This list includes all the basic types, free types, abbreviations, and possible types created for encapsulating multiple inputs and outputs.

The function $JType$ defines the Java type corresponding to each of the used *Circus* types; and $JExp$ translates expressions. The definitions of these functions are simple; for conciseness, we omit them. By way of illustration, the invocation $JType(\mathbb{Z})$ returns `Integer`, and the invocation $JExp(x > y)$ returns `x.intValue() > y.intValue()`.

Table 6.1 presents a summary of the environments that are used throughout the translation strategy. Some of these environments have not already been described; they will be presented and described as we use them.

This section is organised as follows: the rules of translation of processes declarations are presented in Section 6.2.1. Section 6.2.2 presents the translation of the body of basic processes, which is followed by the translation of the CSP actions (Section 6.2.3), and commands (Section 6.2.4). The translation of compound processes is presented in Section 6.2.5. Section 6.2.6 presents how to run the program. The sections that follow extend the strategy by providing means to translate synchronisation channels (Section 6.2.7), the *Circus* indexing operator (Section 6.2.8), generic channels (Section 6.2.9), and multi-synchronised channels (Section 6.2.10). For conciseness, we omit some of the formal definitions of our translation strategy. They can be found in [71].

6.2.1 Processes Declarations

Each process declaration is translated into a class that implements the JCSP interface `jcsp.lang.CSProcess`. For a process P in a project named $proj$, we declare a class `P` that imports the Java utilities package, the basic JCSP package, and all the project packages.

Rule 6.1 $[\![\,\mathbf{process}\ P \mathrel{\widehat=} ParProc]\!]^{ProcDecl} proj =$

```
package proj.processes;
import java.util.*; import jcsp.lang.*;
import proj.axiomaticDefinitions.*;
import proj.typing.*; import proj.util.*;
public class P implements CSProcess {[[ParProc]]^ParProc P }
```

The function $[\![\_]\!]^{ProcDecl}$ takes a *Circus* process declaration and a project name to yield an

Name	Description	Observations
δ	Gives the type of every channel in the system	
ν	For every visible channel within a process, tells if the channel is used as an input, output, or alting channel	Available for each process
ι	For every hidden channel within a process, tells if the channel is used as an input, output, or alting channel	Available for each process
τ	Contains all the types that are used within the system	
λ	Gives the types of every local variable and state component in scope for a given action	Available for each action
ς	For every channel within the system, tells if the channel is used for communication of values or not	Used only for dealing with generic channels
ω	For every channel involved in a multi-synchronisation within the system, gives a function that identifies every process involved in the synchronisation, the number of processes that take part in the synchronisation, and the identity of the process responsible for writing in the synchronisation	Used only for dealing with multi-synchronisation

Table 6.1: Environments used in the Translation Strategy

Java class definition; our rule defines this function. The body of the class is determined by the translation of the paragraphs of P.

As an example, we translate *Register*, *SumClient*, and *Summation* (Figure 2.1 in Chapter 2); the resulting code is in [71] and the code for *Register* is in Figure 6.5 (Page 143). The translation of *Register* is shown below; we omit package and import declarations.

$$\texttt{public class Register implements CSProcess}$$
$$\texttt{\{}[\![\ \textbf{begin}\ \ldots\ \bullet\ \mathit{value} := 0;\ (\mu\, X \bullet \ldots\ )\ \textbf{end}\ ]\!]^{\mathit{ParProc}}\texttt{Register \}}$$

The translation of the body of a parametrised process is captured by the rule presented below.

Rule 6.2 $[\![D \bullet P]\!]^{\mathit{ParProc}} N = (\mathit{ParDecl}\ D)\ (\mathit{VisCDecl}\ \nu)\ (\mathit{HidCDecl}\ \iota)$

$$\begin{array}{l}
\texttt{public}\ N\texttt{(}\mathit{ParArgs}\ D\texttt{,}\mathit{VisCArgs}\ \nu\texttt{) \{} \\
\quad (\mathit{MAss}\ (\mathit{ParDecl}\ D)\ (\mathit{ParArgs}\ D)) \\
\quad (\mathit{MAss}\ (\mathit{VisCDecl}\ \nu)\ (\mathit{VisCArgs}\ \nu)) \\
\quad \mathit{HidCC}\ \iota\ \texttt{\}} \\
\texttt{public void run()\{}\ [\![P]\!]^{\mathit{Proc}}\ \texttt{\}}
\end{array}$$

The process parameters D are declared as attributes: for each $x : T$, the function *ParDecl* yields a declaration `private` $(JType\ T)$ `x;`. The visible channels are also declared as attributes: for each channel c, with use t, $VisCDecl$ gives `private` $(TypeChan\ t)$ `c;`, where $TypeChan\ t$ gives `ChannelInput` for $t = I$, `ChannelOutput` for $t = O$, and `AltingChannelInput` for $t = A$. For $Register$, we have declarations for the channels in the set $RegAlphabet$.

```
private AltingChannelInput store;...; ChannelOutput out;
```

Hidden channels are also declared as attributes, but they are instantiated within the class. We declare them as `Any2OneChannel`, which can be instantiated. The process $Summation$ hides all the channels in the set $RegAlphabet$. For this reason, within the class `Summation` they are declared to be of type `Any2OneChannel`.

The constructor receives the process parameters and visible channels as arguments (the functions $ParArgs\ D$ and $VisCArgs\ \nu$ generate fresh names). The arguments are used to initialise the corresponding attributes using the following expression:

$$MAss\ (ParDecl\ D)\ (ParArgs\ D)$$
$$MAss\ (VisCDecl\ \nu)\ (VisCArgs\ \nu)$$

Furthermore, as explained above, hidden channels are instantiated locally $(HidCC\ \iota)$. In our example, we have the result below.

```
public Register (AltingChannelInput newstore, ...)
    { this.store = newstore; ... }
```

For $Summation$, we have the instantiation of all channels in the set $RegAlphabet$. For instance, `this.store = new Any2OneChannel();` instantiates $store$.

Finally, the method `run` implements the process body translated by $[\![\_]\!]^{Proc}$. In our example, we have `public void run(){`$[\![\ \mathbf{begin}\ \ldots\ \mathbf{end}\ ]\!]^{Proc}$ `}`. For a non-parametrised process, like $Register$, we actually do not use Rule 6.2, but a simpler rule. The difference between the translation of parametrised and non-parametrised processes is only that in a class that corresponds to a parametrised process, we have extra attributes corresponding to parameters.

6.2.2 Basic Processes

Each process body is translated by $[\![\_]\!]^{Proc} : \mathsf{Proc} \rightarrow\!\!\!\!\!\!+\ \mathsf{JCode}$ to an execution of an anonymous inner class that implements `CSProcess`. Inner classes are a Java feature that allows classes to be defined inside classes. The use of inner classes allows compositional translation even in the presence of nameless processes.

Basic processes are translated as follows.

Rule 6.3 $[\![\ \mathbf{begin}\ PPars_1\ \mathbf{state}\ PSt\ PPars_2 \bullet A\ ]\!]^{Proc} =$
`(new CSProcess(){` $(StateDecl\ PSt)$ $([\![PPars_1\ PPars_2]\!]^{PPars})$
`public void run(){`$[\![A]\!]^{Action}$`}}).run();`

The inner class declares the state components as attributes $(StateDecl\ PSt)$. Each action gives rise to a private method ($[\![PPars_1 \ PPars_2]\!]^{PPars}$). The body of `run` is the translation of the main action A. Our strategy ignores any existing state invariants, since they have already been considered in the refinement of the process. The invariants are kept in a *Circus* program just for documentation purposes.

As an example, we present the translation of the body of *Register*. For conciseness, we name its paragraphs *PPars*, and its main action *Main*.

(new CSProcess(){ private Integer value; $[\![PPars]\!]^{PPars}$
public void run() {$[\![Main]\!]^{Action}$} }).run();

The function $[\![\_]\!]^{PPars} : \mathsf{PPar}^* \rightarrowtail \mathsf{JCode}$ translates the paragraphs within a *Circus* process, which can either be axiomatic definitions, or (parametrised) actions. The translation of an axiomatic definition $v : T \mid v = e$ is the following method

private $(JType\ T)$ v(){return $(JExp\ e)$;}

Since the paragraphs of a process p can only be referenced within p, the method is declared `private`. We omit the relevant rule, and a few others in the sequel, for conciseness.

Both parametrised actions and non-parametrised actions are translated into private methods. However, the former requires that the parameters are declared as arguments of the new method. The reason for the method to be declared `private` is the same as that discussed above for the axiomatic definitions.

Rule 6.4 $[\![N \mathrel{\widehat=} (D \bullet A)\ PPars]\!]^{PPars} =$
private void N($ParArgs\ D$){$[\![A]\!]^{Action}$}$[\![PPars]\!]^{PPars}$

The function *ParArgs* declares an argument for each of the process parameters. The body of the method is defined by the translation of the action body.

For instance, the translation of action *RegCycle* generates the following Java code. We use *body* to denote the body of the action.

$[\![RegCycle \mathrel{\widehat=} body]\!]^{PPars} =$ private void RegCycle(){$[\![body]\!]^{Action}$}

The function $[\![\_]\!]^{Action} : \mathsf{Action} \rightarrowtail \mathsf{JCode}$ translates CSP actions and commands.

6.2.3 CSP Actions

In the translation of each action, the environment λ is used to record state components and the local variables in scope in the translation of parallel and recursive actions. For each variable and state component, λ maps its name to its type. As we did for processes, we have channel environments ν and ι to store information about how each channel is used.

The translations of *Skip* and *Stop* use basic JCSP classes: *Skip* is translated into the Java code `(new Skip()).run();`, and *Stop* is translated to `(new Stop()).run();`. *Chaos* is translated to an infinite loop `while(true){};`, which is a valid refinement of *Chaos*. For input communications, we declare a new variable whose value is read from the channel. A cast is needed, since the type of the objects transmitted through the channels is `Object`; we use the channel environment δ to determine the type to which the object should be cast.

Rule 6.5 $[\![\, c?x \rightarrow Act \,]\!]^{Action} =$ `{` t `x = (`t`)c.read();` $[\![Act]\!]^{Action}$`}`
where $t = JType(last\ (snd\,(\delta\ c)))$.

For instance, the communication $add?newValue$ used in the action $RegCycle$ is translated to `Integer newValue = (Integer)add.read();`

An output communication is easily translated as follows.

Rule 6.6 $[\![\, c!x \rightarrow Act \,]\!]^{Action} =$ `c.write(x);` $[\![\, Act \,]\!]^{Action}$

For synchronisation channels, we need to know whether it is regarded as an input or an output channel; this information is retrieved either from ν or ι.

Rule 6.7 $[\![c \rightarrow Act\,]\!]^{Action} =$ `c.read();`
provided $\nu\ c \in \{\, I, A \,\} \vee \iota\ c \in \{\, I, A \,\}$ ❑

Rule 6.8 $[\![c \rightarrow Act\,]\!]^{Action} =$ `c.write(null);`
provided $\nu\ c = O \vee \iota\ c = O$ ❑

For example, in the process $SumClient$, the action $reset \rightarrow Sum(n)$ is translated to the Java code `reset.write(null);`, followed by the translation of $Sum(n)$. Within $Register$, the translation of $reset$ is `reset.read();`. The difference is because $reset$ is an output channel for $SumClient$, and an input channel for $Register$.

Sequential compositions are translated to Java sequential compositions.

Rule 6.9 $[\![\, A_1;\ \ldots\ ;\ A_n \,]\!]^{Action} = [\![\, A_1 \,]\!]^{Action}$`;` $\ldots$ `;` $[\![\, A_n \,]\!]^{Action}$

The translation of external choice uses the corresponding `Alternative` JCSP class; all the initial visible channels involved take part.

Rule 6.10 $[\![A_1 \,\square \ldots \square\, A_n]\!]^{Action} =$

`Guard[] g = new Guard[]{`$ICAtt\ A_1$`,...,`$ICAtt\ A_n$`};`
`final Alternative alt = new Alternative(g);`
$(DeclCs\ (ExIC\ A_1)\ 0)\ \ldots\ (DeclCs\ (ExIC\ A_n)\ (\#(ExIC\ A_{n-1})))$
`switch(alt.select()){`$Cases\ (ExIC\ A_1)\ A_1 \ldots\ Cases\ (ExIC\ A_n)\ A_n$`}`

provided $A_1,\ \ldots,\ A_n$ *are not guarded actions* $g_i\ \&\ A_i$. ❑

In this chapter, $\#s$ stands for the length of the sequence s.

In Figure 6.2 we present the translation of the body of $RegCycle$. It declares an array containing all initial visible channels of the choice (1). The function $ICAtt$ returns a comma-separated list of all initial visible channels of an action; informally, these are the

```
Guard[] guards = new Guard[]{store,add,result,reset};           (1)
final Alternative alt = new Alternative(guards);                (2)
final int C_STORE = 0; ... ; final int C_RESET = 3;             (3)
switch(alt.select())                                            (4)
    { case C_STORE:{...} break; ...; case C_RESET:{...} break; } (5)
```

Figure 6.2: Example of External Choice Translation - Action *RegCycle*(Page 23)

first channels through which the action is prepared to communicate. The array is used in the instantiation of the `Alternative` process (2). Next, an `int` constant is declared for each channel (3). The function *DeclCs* returns a semicolon-separated list of `int` constant declarations. The first constant is initialised with 0, and each subsequent constant with the previous constant incremented by one. Finally, a choice is made, and the chosen action executed. We use a `switch` block (4); the body of each `case` is the translation of the corresponding action (5); the function *Cases* takes the initial visible channel as argument (*ExIC*).

For guarded actions $\Box\, i \bullet g_i \,\&\, A_i$, we have to declare an array `g` of booleans $\mathit{JExp}\ g_i$. We use this array in the selection `alt.select(g)`. Each unguarded action A_i can be easily refined to $\mathit{true} \,\&\, A_i$. If the guards are mutually exclusive, we can apply a different rule to obtain an `if-then-else`. This simplifies the generated code, and does not require the guarded actions to be explored in the translation of the external choice.

The translation of an internal choice chooses a random number between 1 and n. It uses the static method `generateNumber` of class `RandomGenerator`. Finally, it uses a `switch` block to choose and run the chosen action.

Rule 6.11 $[\![A_1 \sqcap \ldots \sqcap A_n]\!]^{\mathit{Action}} =$
`switch(RandomGenerator.generateNumber(1,n))`
`{case 1:{`$[\![A_1]\!]^{\mathit{Action}}$`}break;`...`case n:{`$[\![A_n]\!]^{\mathit{Action}}$`}break;}`

To translate a parallel composition, we define an inner class for each parallel action, because the JCSP `Parallel` constructor takes an array of processes as argument. To deal with the partition of the variables, we use auxiliary variables to make copies of each state component and local variable in scope. The body of each branch is translated and each reference to state components or local variables is replaced with its copy. After the parallel composition, we merge the values of the variables in each partition. The copies are initialised in the constructor of each parallel action. Their initial values are given to the constructor as arguments.

The names of the inner classes are defined in the translation. To avoid clashes, we use a fresh index *ind* in the name of inner classes and local variables copies. In the following rule, *LName* and *RName* stand for the names of the classes that implement A_1 and A_2. We omit *RName*, which is similar to *LName*.

The function *DeclLcCopies* declares one copy of each state component and local variable in scope; the initial values are taken by the constructor (*LcCopiesArgs*). In the body of the constructor, the function *ILcCopies* initialises the copies with the corresponding values received as argument. The body of the method `run` is the translation of the action. The function *RenVars* is used to replace occurrences of the state components and variables in scope with their copies.

After the conclusion of the declaration of the inner class *LName*, we create an object of *LName*. A similar approach is taken in the translation of A_2 to *RName* and an object creation. The next step is to run the parallel composition. Afterwards, a merge retrieves the final values of the state components and the variables in scope from their copies (*MergeVars*).

Rule 6.12 $[\![A_1 \, [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \, A_2]\!]^{Action} =$

```
class LName implements CSProcess {
    (DeclLcCopies λ ind L)
    public LName((LcCopiesArg λ)) { ILcCopies λ ind L }
    public void run() {
        RenVars [[A1]]^Action (ListFirst λ) ind L } }
CSProcess l_ind = new LName(JList (ListFirst λ));
//class RName declaration, process r_ind instantiation
CSProcess[] procs_ind = new CSProcess[]{ l_ind,r_ind };
(new Parallel(procs_ind)).run();
(MergeVars LName ns1 ind L) (MergeVars RName ns2 ind R)
```

where *LName* = `ParLBranch_`*ind* *and RName* = `ParRBranch_`*ind*

For instance, we present the translation of $x := 0 \, [\![\{x\} \mid \varnothing \mid \{y\}]\!] \, y := 1$ in Figure 6.3. We consider that the action occurs within a process with one state component $x : \mathbb{Z}$, and that there is one local variable $y : \mathbb{Z}$ in scope.

The class `ParLBranch_0` has two attributes: one corresponding to the state component x (2) and one corresponding to the local variable y (3); their initial values are received in the constructor (4). The body of the method `run` (8) replaces all the occurrences of `x` by its copy `aux_l_x_0`. This concludes the declaration of the class `ParLBranch_0`, which is followed by the creation of an object `l_0` of this class (9). For conciseness, we omit the declaration of the class related to the right-hand side of the parallel composition (10). Its declaration, however, is very similar to the left-hand side: the copies of the state component x and the local variable y are declared and initialised as in class `ParLBranch_0`; the body of method `run` is the assignment `aux_r_y_0 = new Integer(1);`. Finally, after running the parallel composition (11,12), the final value of `x` is that of its left branch copy (13), and the final value of `y` is that of its right branch copy (14).

If we have a *Circus* action invocation, all we have to do is to translate it to a method call. If no parameter is given, the method invocation has no parameters. However, if any parameter is given, we use a Java expression corresponding to each parameter in the method invocation. In our example, $Sum(n)$ and $Sum(n-1)$ translate to `Sum(n);` and `Sum(new Integer(n.intValue()-1));`.

We also use inner classes to declare the body of recursions. As for parallel composition, this requires the use of copies of state components and local variables, which are declared

```
class ParLBranch_0 implements CSProcess {
    public Integer aux_l_x_0;
    public Integer aux_l_y_0;
    public ParLBranch_0(Integer x, Integer y) {
        this.aux_l_x_0 = x;
        this.aux_l_y_0 = y;
    }
    public void run() { aux_l_x_0 = new Integer(0); } }
CSProcess l_0 = new ParLBranch_0(x,y);
\* Right-hand side of the parallel composition *\
CSProcess[] procs_0 = new CSProcess[]{l_0,r_0};
(new Parallel(procs_0)).run ();
x = ((ParLBranch_0)procs_0[0]).aux_l_x_0;
y = ((ParRBranch_0)procs_0[1]).aux_r_y_0;
```

Figure 6.3: Example of Parallel Operator Translation

as attributes of the inner class, and initialised in its constructor with the values given as arguments. The `run` method of this new inner class executes the body of the recursion, instantiates a new object of this class, where the recursion occurs, and executes it.

Rule 6.13 $[\![\, \mu\, X \bullet A(X) \,]\!]^{Action} =$

`class I_`ind `implements CSProcess {`
  $DeclLcCopies\ \lambda\ ind\ L$
  `public I_`ind`(`$LcCopiesArg\ \lambda$`) {` $ILcCopies\ \lambda\ ind\ L$ `}`
  `public void run(){`
    $RenVars\ [\![A(RunRec\ ind)]\!]^{Action}\ (\mathrm{dom}\ \lambda)\ ind\ L$ `} };`
$(RunRec\ ind)$

The function *RunRec* instantiates a recursion process, invokes its `run` method, and finally retrieves the final values of the state components and local variables in scope. For the same reason as for the translation of parallel composition, we use a fresh index in the name of the inner class created for the recursion.

For instance, in Figure 6.4, we present the translation of the main action of process *Register*. First, we initialise `value` with 0 (1). Next, we declare the class `I_0`, which implements the recursion. It has a copy of the state component `value` as its attribute (3), which is initialised in the constructor (4). The method `run` calls the method `RegCycle` (6), instantiates a new recursion (7), executes it (7), and retrieves the final value of the local copy of `value` (8); this concludes the declaration of the recursion class. Next, we instantiate an object of this class, and execute it (9). Finally, we retrieve the final `value` (10).

In order to reuse the previous definitions, the translation of parametrised unnamed action invocations also makes use of inner classes. Since inner classes cannot access the

```
value:=new Integer(0);
class I_0 implements CSProcess {
    public Integer aux_l_value_0;
    public I_0(Integer value){ this.aux_l_value_0 = value; }
    public void run() {
        RegCycle();
        I_0 i_0_0 = new I_0(aux_l_value_0); i_0_0.run();
        aux_l_value_0 = i_0_0.aux_l_value_0; } };
I_0 i_0_0 = new I_0(value); i_0_0.run();
value = i_0_0.aux_l_value_0;
```

Figure 6.4: Example of Recursion Translation

attributes corresponding to the state components and local variables in scope, each one of them have a corresponding copy as an attribute of the new class. The action parameters are also declared as attributes of the new class; both the attributes corresponding to the copies of the state components and local variables, and parameters are initialised within the class constructor with the corresponding values given as arguments. The `run` method of the new class executes the parametrised action. However, the references to the local variables are replaced by references to their copies. Next, the translation creates an object of the class with the given arguments, and calls its `run` method. Finally, it restores the values of the local variables.

The translation of iterated sequential composition is presented below.

Rule 6.14 $[\![\mathbin{⨟} x_1 : T_1; \ldots; x_n : T_n \bullet Act]\!]^{Action} =$
InstActions `pV_`*ind* $(x_1 : T_1; \ldots; x_n : T_n)$ *Act ind*
`for(int i = 0; i < pV_`*ind*`.size(); i++)`
`{ ((CSProcess)pV_`*ind*`.elementAt(i)).run(); }`

The function *InstActions* declares an inner class `I_`*ind* that implements the action *Act* parametrised by the indexing variables. Then, it creates a vector `pV_`*ind* of actions using a nested loop over the possible values of each indexing variable: for each iteration, an object of `I_`*ind* is created using the current values of the indexing variables, and stored in `pV_`*ind*. Finally, each action within `pV_`*ind* is executed in sequence.

The translation of iterated internal choice uses the `RandomGenerator` to choose a value for each indexing variable. Then, it instantiates an action using the chosen values, and runs it.

6.2.4 Commands

Single assignments are directly translated to Java assignments.

Rule 6.15 $[\![x := e]\!]^{Action} =$ `x = (`*JExp e*`);`

For multiple assignments, however, we have two cases. Assignments in which no expression in the right-hand side of the assignment mention any variable in its left-hand side are implemented simply as a sequence of each single assignment.

Rule 6.16 $[\![x_1,\ldots,x_n := e_1,\ldots,e_n]\!]^{Action} =$ `x_1=`$(JExp\ e_1)$`;` $\ldots$`;` `x_n=`$(JExp\ e_n)$`;`
provided $\{x_1,\ldots,x_n\} \cap (FV(e_1) \cup \ldots \cup FV(e_n)) = \varnothing$ ❑

Otherwise, we create a copy of every variable involved in the assignment, and use these copies in the assignment. The types of the copy variables are the same as the original variables; they are retrieved from the variables environment λ.

Rule 6.17 $[\![x_1,\ldots,x_n := e_1,\ldots,e_n]\!]^{Action} =$
$(JType\ (\lambda\ x_1))$ `aux_`ind`_x_1 = `$(JExp\ e_1)$`;`
$\ldots$`;`
$(JType\ (\lambda\ x_n))$ `aux_`ind`_x_n = `$(JExp\ e_n)$`;`
`x_1=aux_`ind`_x_1;` $\ldots$`;` `x_n=aux_`ind`_x_n;`

provided $\{x_1,\ldots,x_n\} \cap (FV(e_1) \cup \ldots \cup FV(e_n)) \neq \varnothing$ ❑

Variable declarations only introduce the declared variables in scope.

Rule 6.18 $[\![\,\mathbf{var}\ x_1 : T_1;\ldots;x_n : T_n \bullet Act]\!]^{Action} =$
`{` $(JType\ T_1)$ `x_1;`$\ldots$`;` $(JType\ T_n)$ `x_n;` $[\![Act]\!]^{Action}$ `}`

Alternations(**if**_**fi**) are translated to `if-then-else` blocks; possible nondeterminism is removed by choosing the first *true* guard. If none of the guards is *true*, the action behaves like *Chaos* (`while(true){}`).

Rule 6.19 $[\![\,\mathbf{if}\ g_1 \rightarrow A_1 \,\Box \ldots \Box\, g_n \rightarrow A_n\ \mathbf{fi}\,]\!]^{Action} =$
`if(`$JExp\ g_1$`){`$[\![A_1]\!]^{Action}$`}...else if(`$JExp\ g_n$`){`$[\![A_n]\!]^{Action}$`}`
`else { while(true){} }`

At this point, we are able to translate basic processes. By way of illustration, Figure 6.5 presents the complete translation of process *Register*.

6.2.5 Compound Processes

We now concentrate in the translation of the processes that are defined in terms of other processes. At this stage, we are actually translating the body of some process (Figure 6.1). This means we are translating the body of its method `run`.

For a single process name N, we must instantiate the process `N`, and then, invoke its `run` method. The visible channels of the process are given as arguments to the process constructor. The function *ExtChans* returns a list of all channel names in the domain of the environment ν.

Rule 6.20 $[\![\,N\,]\!]^{Proc} =$ `(new N(`$ExtChans\ \nu$`)).run();`

The invocation of unnamed parametrised processes is translated to a new inner class.

```
// Package declaration and imports (See Rule 6.1)
public class Register implements CSProcess {
  private AltingChannelInput store; private AltingChannelInput add;
  private AltingChannelInput result; private AltingChannelInput reset;
  private ChannelOutput out;
  public Register (AltingChannelInput newstore, AltingChannelInput newadd,
                   AltingChannelInput newresult,
                   AltingChannelInput newreset, ChannelOutput newout) {
    this.store = newstore; this.add = newadd; this.result = newresult;
    this.reset = newreset; this.out = newout;  }
  public void run(){
    (new CSProcess(){
      private Integer value;
      private void RegCycle(){
        Guard[] guards = new Guard[]{store,add,result,reset};
        final Alternative alt = new Alternative(guards);
        final int C_STORE = 0; final int C_ADD = 1;
        final int C_RESULT = 2; final int C_RESET = 3;
        switch(alt.select()) {
          case C_STORE:
            { { Integer newValue = (Integer)store.read();
                value = newValue; } } break;
          case C_ADD:
            { { Integer newValue = (Integer)add.read();
                value = new Integer(value.intValue() +
                                    newValue.intValue()); } } break;
          case C_RESULT:
            { result.read(); out.write(value);
            (new Skip()).run(); } break;
          case C_RESET:
            { reset.read(); value = new Integer(0); } break; } }
      public void run() {
        value = new Integer(0);
        class I_0 implements CSProcess {
          public Integer aux_l_value_0;
          public I_0(Integer value) { this.aux_l_value_0 = value; }
          public void run () {
            RegCycle(); I_0 i_0_0 = new I_0(aux_l_value_0);
            i_0_0.run(); aux_l_value_0 = i_0_0.aux_l_value_0; } }
          I_0 i_0_0 = new I_0(value); i_0_0.run();
          value = i_0_0.aux_l_value_0; } }).run(); } }
```

Figure 6.5: Translation of Process *Register* (Figure 2.1, Page 23)

It runs the parametrised process instantiated with the given arguments. The new class name is also indexed by a fresh *ind* to avoid clashes.

The sequential composition of processes is also easily translated to the sequential execution of each process.

Rule 6.21 $[\![\, P_1 ;\ \ldots ;\ P_n \,]\!]^{Proc} = [\![P_1]\!]^{Proc}\ ;\ \ldots\ ;\ [\![P_n]\!]^{Proc}$

External choice has a similar solution to that presented for actions. The idea is to create an alternative in which all the initial channels of both processes, that are not hidden, take part. However, all auxiliary functions used in the previous definitions apply to actions. All we have to do is use similar functions that take processes into account.

As the internal choice for actions, the internal choice $P_1 \sqcap \ldots \sqcap P_n$ for processes randomly chooses a process, and then executes it. Its definition is very similar to the corresponding one for actions.

The translation of parallel operator executes a `Parallel` process. This process executes all the processes that are elements of the array given as argument to its constructor in parallel. In our case, this array has only two elements: each one corresponds to a process of the parallel composition. Furthermore, the translation of parallel composition of processes does not have to take into account variable partitions.

Rule 6.22 $[\![P_1 \,[\![\, cs \,]\!]\, P_2]\!]^{Proc} =$

```
(new CSProcess(){
    public void run(){
        new Parallel(
            new CSProcess[]{
                new CSProcess(){ public void run(){[[P1]]^Proc} },
                new CSProcess(){ public void run(){[[P2]]^Proc} }
            }).run();
    }
}).run();
```

It is important to notice that, when using JCSP, the intersection of the alphabets determines the synchronisation channels set. For this reason, *cs* may be ignored.

The renaming operation $P[x_1, \ldots, x_n := y_1, \ldots, y_n]$ is translated by replacing all the `x_is` by the corresponding `y_is` in the translated Java code of P.

As for actions, the iterated operators are translated using `for` loops. The same restrictions on the type of the indexing variables apply for processes. The first iterated operator on processes is the sequential composition $⨾$. As for actions, we use an auxiliary function to create a vector of processes, and execute in sequence each process within this vector. The iterated internal choice chooses a value for each indexing variable, and runs the process with the randomly chosen values for the indexing variables.

The translation of iterated parallel composition of processes are simpler than that of actions, since we do not need to deal with partitions of variables in scope.

Rule 6.23 $[\![\, \|[\, cs \,]\| \; x_1 : T_1; \; \ldots; \; x_n : T_n \bullet P]\!]^{Proc} =$

```
(new CSProcess(){
    public void run(){
      InstProcs pV_ind (x1 : T1; ...; xn : Tn) P ind
      CSProcess[] pA_ind = new CSProcess[pV_ind.size()];
      for (int i = 0; i < pV_ind.size(); i++)
          { pA_ind[i] = (CSProcess)pV_ind.get(i); }
      (new Parallel(pA_ind)).run(); } }).run();
```

It uses the function *InstProcs* to instantiate a vector `pV_`*ind* containing each of the processes obtained by considering each possible value of the indexing variables. Then, it transforms this `pV_`*ind* into an array `pA_`*ind*, which is given to the constructor of a `Parallel` process. Finally, we run the `Parallel` process.

6.2.6 Running the program

The function $[\![\_]\!]^{Program}$ summarises our translation strategy. Besides the *Circus* program, this function also receives a project name, which is used to declare the package for each new class. It declares the class that encapsulates all the axiomatic definitions (*DeclAxDefCls*), and translates all the declared processes.

Rule 6.24 $[\![Types \;\; AxDefs \;\; ChanDecls \;\; ProcDecls]\!]^{Program} proj =$
$(DeclAxDefCls \;\; proj \;\; AxDefs) \, ([\![ProcDecls]\!]^{ProcDecls} proj)$

In order to generate a class with a `main` method, which can be used to execute a given process, we use the function $[\![\_]\!]^{Run}$. This function is applied to a *Circus* process name and a project name. It creates a Java class named `Main`, which is created in the package *proj*. After the package declaration, the class imports the packages `java.util`, `jcsp.lang`, and all the packages within the project. The method `main` is defined as the translation of the given process.

For instance, in order to run the process *Summation*, we have to apply the function $[\![\_]\!]^{Run}$ to this process and give the project name `sum` as argument. This application results in the following Java code.

```
package sum;
import jcsp.lang.*;
import summation.typing.*; import summation.processes.*;
import summation.util.*;
public class Main {
    public static void main(String args[]) {
        (new CSProcess(){
            public void run(){ (new Summation()).run(); } }).run();
    }
}
```

The execution of this class executes the process *Summation*.

6.2.7 Synchronisations

In this section, we extend the types of communications considered in our strategy; we deal with communication events of the form $\mathsf{N.Expression}$. Our strategy implements synchronisation using arrays of channels. Throughout this section, we illustrate our definitions using the channel *gasDischarged*, which is used in Chapter 5 to indicate that gas has been discharged in a particular area.

channel *gasDischarged* : *AreaId*

For example, the synchronisation *gasDischarged*.0, which represents a gas discharge in area 0, is implemented as `gasDischarged[0]`, the 0th element in the array *gasDischarged*. Basically, each synchronisation *.exp* is implemented as an additional dimension in an array of channels. In order to simplify our definitions, we consider that the uses of such channels first declare possible synchronisation of the form $\mathsf{.Exp}$, and finally possible communications of the form $\mathsf{?N}$ or $\mathsf{!Exp}$. For the purpose of characterising the kind of communications contemplated by our strategy, our definitions of Comm and $\mathsf{CParameter}$ are altered below.

$$
\begin{array}{lll}
\mathsf{Comm} & ::= & \mathsf{N\,.Exp^{*}\,CParameter} \\
\mathsf{CParameter} & ::= & \mathsf{?N} \ \mid \mathsf{!Exp} \mid \mathsf{.Exp}
\end{array}
$$

Our strategy still constrains the channels to have only one input or output value. Multiple inputs and outputs must be encapsulated in Java objects.

Another important constraint is that if a channel c is used in a synchronisation of the form $\mathsf{N.Exp}$, it must be declared as **channel** $c : T$, where T is finite. This constraint arises from the fact that our strategy uses arrays of channels for representing synchronisation events. In order to determine the dimension of the arrays, we use the maximum and the minimum values of the type of the channel. In the case of infinite types, we would not be able to calculate the dimension of the arrays.

A very important change in this extension is the use of a new channel environment $\varsigma\colon \mathsf{N} \rightarrow SC$. It maps each channel used within the system to a value of type SC, which indicates if the channel is a communication channel (C), or a synchronisation channel (S). In our example, the channel *gasDischarged* is mapped to the value S.

The changes in the translation strategy are concerned with the declaration, instantiation, and use of these channels. Therefore, all the previously defined functions that are used to translate these aspects of the ***Circus*** programs must be redefined.

First, the function *VisCDecl* is changed in order to deal with the possibility of channel array declarations. Besides the type and the name of the channel, this function, and others that follow, use an auxiliary function *ArrayDimSync* in order to check the dimension of the array of channels that implements the given channel. If this dimension is equal to zero, the channel is implemented in the same way as previous definitions: a single channel.

The function *ArrayDimSync* receives two arguments: the type of the channel, *types*, and a value *sc* of type SC indicating if the channel is a synchronisation or a communication channel.

If the channel is untyped, the list *types* is a singleton with the element *Sync*. In this case, the dimension is zero. However, the dimension for typed channels is as follows. If no inputs and outputs are involved in the communications through a channel (S), the dimension of the array is equal to the size of the *types* list. Otherwise, if the channel is a communication channel (C), the last type indicates the type that is communicated, and therefore, we remove one from the final array dimension. We return as many `[]`'s as the dimension we calculated for the array. We use the notation $(\texttt{code})^n$ to represent n repetitions of `code`; if $n \leq 0$, $(\texttt{code})^n$ is the empty string ϵ.

Definition 6.1

$$
\begin{aligned}
&\mathit{ArrayDimSync}\ \mathit{types}\ \mathit{sc} = \mathbf{let}\ \mathit{dim} = \mathbf{if}\ (\mathit{types} = [\mathit{Sync}])\ \mathbf{then}\ 0\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ if}\ (\mathit{sc} = C)\ \mathbf{then}\ \#\mathit{types} - 1\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \#\mathit{types}\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \texttt{[]}^{\,\mathit{dim}}
\end{aligned}
$$

A local definition is used to make the definition more concise: we use the notation **let** $n = e$ **in** p to represent the substitution in p of n by e.

For channel *gasDischarged*, we have that $\mathit{types} = [\mathit{AreaId}]$, $\mathit{sc} = S$. The application of function *ArrayDimSync* with these arguments returns the string `[]`; in the process *FireControl*, the function *VisCDecl* returns `private ChannelInput[] gasDischarged` for the declaration of channel *gasDischarged*.

As for the used channels, we redefine the function *HidCDecl*. The definition of the dimension of possible arrays of channels is the same as for the visible channels. However, we declare the channels as `Any2OneChannel` channels, since they are instantiated within this process. The redefinition of function *VisCArgs* is very similar to the original one. However, it also takes into account the existence of possible channel arrays, using the auxiliary function *ArrayDimSync*.

If a hidden channel is not declared as an array the channel is instantiated as a `Any2OneChannel` channel. Otherwise, we use the auxiliary function *InstArraySync* to instantiate the channel as an array of channels.

The function *InstArraySync* instantiates an array of channels. It receives the *types* used in the declaration of the channel, and a value *sc* of type *SC*, indicating whether the channel is used for inputs and outputs or not. If we have only one type in the list of types used in the channel declaration, we use the function *BaseCase* to declare either a channel instantiation (C), or an array of channels creation (S). Otherwise, we instantiate an array of channels with dimension defined by the function *ArrayDimSync*. The function *InstArraySync* is used in order to instantiate each of the elements in the array.

Definition 6.2

$$
\begin{aligned}
&\mathit{InstArraySync}\ \mathit{types}\ \mathit{sc} =\\
&\quad \mathbf{let}\ \mathit{type} = (\mathit{head}\ \mathit{types}), \mathit{dim} = (\mathit{ArrayDimSync}\ \mathit{types}\ \mathit{sc})\\
&\quad \mathbf{in\ if}\ (\#\mathit{types} = 1)\ \mathbf{then}\ \mathit{BaseCase}\ \mathit{type}\ \mathit{sc}\\
&\qquad \mathbf{else}\ \texttt{new Any2OneChannel}\ \mathit{dim}\\
&\qquad\quad \texttt{\{}\ \mathit{TypeInstSync}\ \mathit{types}\ \mathit{sc}\\
&\qquad\qquad\qquad (\mathit{Max}(\mathit{JType}(\mathit{type}))) - (\mathit{Min}(\mathit{JType}(\mathit{type}))) + 1\ \texttt{\}}
\end{aligned}
$$

Our example falls in the first case: we have that *BaseCase AreaId S* instantiates this channel. If the channel is a communication channel (C), the function *BaseCase* instantiates a single `Any2OneChannel()` channel; otherwise (S), it instantiates an array of channels with the number of elements equal to the number of possible values of the type T given as argument using the expression below.

$$\texttt{Any2OneChannel.create((}Max(JType(T))\texttt{)-(}Min(JType(T))\texttt{)+1)}$$

In JCSP, the static function `create` creates an array of `Any2OneChannel`s. The functions *Max* and *Min* return the code `MAX_T` and `MIN_T`, which represent the maximum and the minimum values in the Java type T given as argument, respectively. The channel *gasDischarged* is instantiated as follows.

```
this.gasDischarged =
    Any2OneChannel.create(MAX_AREA_ID - MIN_AREA_ID + 1);
```

The function *TypeInstSync* invokes the function *InstArraySync* for the other type expressions used in the channel declaration (*tail types*) for each element in the current type (*head types*). This is done by invoking the function *TypeInstSync*, giving the number of elements in the type that is in the head of the list *types* as argument. The function *tail*, as expected, returns the tail of a given list.

Definition 6.3

$$\begin{aligned}
&TypeInstSync\ types\ sc\ 1 = InstArraySync\ (tail\ types)\ sc\\
&TypeInstSync\ types\ sc\ n = InstArraySync\ (tail\ types)\ sc\texttt{,}\\
&\qquad\qquad\qquad\qquad\qquad\quad TypeInstSync\ types\ sc\ (n-1)
\end{aligned}$$

Most of the translation of actions remains the same; only those that are concerned with communications and external choice must be extended. For a given input channel c, the type of communicated value (*commType*) is given by the Java type of the last element in the list (*last*) of types of c. This is the type used to declare an input variable, if needed. Each synchronisation $.i$ is translated to an access of the i-th element in an array of channels.

Rule 6.25 $[\![c\ .e_0\ \ldots\ .e_m?x \to Act]\!]^{Act} =$

$$\begin{aligned}
&\textbf{let}\ commType = JType(last\ (snd\ (\delta\ c)))\\
&\textbf{in}\ \texttt{\{}\ commType\ \texttt{x = (}commType\texttt{)c[}JExp\ e_0\texttt{]}\ldots\texttt{[}JExp\ e_n\texttt{].read();}\\
&\qquad [\![Act]\!]^{Act}\texttt{\}}
\end{aligned}$$

Given a triple (a, b, c), we have that the functions *fst*, *snd*, and *trd* return a, b, and c, respectively.

An output still writes to the channel. Again, we use the function *JExp* in order to access the correct element in the array of channels.

Rule 6.26 $[\![c\ .e_0\ \ldots\ .e_m!x \to Act]\!]^{Act} =$

$$\texttt{c[}JExp\ e_0\texttt{]}\ldots\texttt{[}JExp\ e_n\texttt{].write(}JExp\ x\texttt{);}\ \ [\![Act]\!]^{Act}$$

Finally, synchronisation channels simply read from a channel or write a null value to the corresponding channel.

Rule 6.27 $[\![c\ .e_0\ \ldots\ .e_m \rightarrow Act]\!]^{Act} =$ `c[`$JExp\ e_0$`]`$\ldots$`[`$JExp\ e_n$`].read(); `$[\![Act]\!]^{Act}$
provided $\nu\ c \in \{\, I, A \,\} \lor \iota\ c \in \{\, I, A \,\}$ ❑

Rule 6.28 $[\![c\ .e_0\ \ldots\ .e_m \rightarrow Act]\!]^{Act} =$ `c[`$JExp\ e_0$`]`$\ldots$`[`$JExp\ e_n$`].write(); `$[\![Act]\!]^{Act}$
provided $\nu\ c = O \lor \iota\ c = O$ ❑

In the case of external choice, we must redefine some of the auxiliary functions to take into account the existence of arrays of channels; each channel in an array of channels is considered as a different visible channel.

The function *ICAtt*, which is used in the declaration of the array of channels that take part in the external choice, is redefined in order to take into account possible synchronisation values in the channels. This means that synchronisations of the form $c.x_0 \ldots .x_m$ are directly translated to `c[(`$JExp\ x_0$`)]...[(`$JExp\ x_m$`)]`.

The function *ExIC*, that extracts the initial channels of a given action, returns a list of pairs. For each initial visible channel of the given action, it includes a new pair in this list: the first element is the channel (possibly with its synchronisation values) and the second element is a predicate that represents its guard.

The function *DeclCs* is used to declare one constant for each channel that takes part in the external choice. It returns a semicolon-separated list of `int` constant declarations. These constants make the resulting code easier to understand. Its definition, however, takes into account the possible existing arrays of channels. Therefore, for each channel $c.x_0 \ldots .x_m$, we have a new constant declaration as follows.

```
final int CONST_C_X_0_...X_m = n;
```

Finally, the redefined function *Cases*, which returns a sequence of Java `case` blocks, one for each initial channel in a given channel list, also returns a different case block for each element in an array of channel that takes part in the external choice.

As an example, we have the external choice below. For simplicity, we consider that the channels a and b are declared as **channel** $a, b : \{0..1\}$.

$$(a.0 \rightarrow Skip) \,\Box\, (a.1 \rightarrow Skip) \,\Box\, (b.0 \rightarrow Stop) \,\Box\, (b.1 \rightarrow Stop)$$

First, we declare the array that contains all the visible channels within the action, and an `Alternative` on this array. Notice that each element of the channel arrays are considered as different visible channels.

```
Guard[] guards = new Guard[]{a[0],a[1],b[0],b[1]};
final Alternative alt = new Alternative(guards);
```

Then, all the constants that are used in the `switch` block are declared. They identify each possible choice that can be made by the `select` method.

```
final int CONST_A_0 = 0; final int CONST_A_1 = 1;
final int CONST_B_0 = 2; final int CONST_B_1 = 3;
```

Finally, we have a `switch` block. For each value that can be returned by the method `select` invocation, we have a `case`, which reads from the corresponding channel, and then behaves like the translation of the corresponding action.

```
switch(alt.select(g)) {
    case CONST_A_0:
        { a[0].read(); (new Skip()).run(); } break;
    case CONST_A_1:
        { a[1].read(); (new Skip()).run(); } break;
    case CONST_B_0:
        { b[0].read(); (new Stop()).run(); } break;
    case CONST_B_1:
        { b[1].read(); (new Stop()).run(); } break;
}
```

This concludes the translation of our example.

With a translation strategy for synchronisation channels of the form $c.e$ we can easily deal with the *Circus* indexing operator, as we present in the sequel.

6.2.8 Indexing Operator

An indexed process can be seen as a kind of parametrised process. The difference, however, is that a syntactic substitution on the channels, as defined in Chapter 3, is made. It is very important to notice that the creation of the channels environment already takes into account the indexed processes. So, the channels implicitly created by the indexed operator are already within the channel environment. For instance, consider the following channel environment $\delta \mathrel{\widehat=} \{c \mapsto ([\,], [\mathbb{Z}])\}$ that has a channel c of type $\mathbb{Z}$. In the translation of the process $x : T \odot \mathit{Proc}$, we consider that the environment δ is extended to $\delta \mathrel{\widehat=} \{c \mapsto ([\,], [\mathbb{Z}]), c\_x \mapsto ([\,], [T, \mathbb{Z}])\}$.

The renaming in the channels within a given indexed process $\mathit{Decl} \odot \mathit{Proc}$ is reflected in the way the channels are instantiated, referenced, and used. An indexed process $x_1 : T_1; \ldots; x_n : T_n \odot \mathit{Proc}$ is translated as the following parametrised process.

$$x_1 : T_1; \ldots; x_n : T_n \bullet \mathit{Proc}$$

However, for every channel c used within the process, we replace every reference to c, by a reference to $c\_x\_1 \ldots x\_n.x_1. \ \ldots \ .x_n$.

Rule 6.29 $[\![\, x_1 : T_1; \ldots; x_n : T_n \odot \mathit{Proc} \,]\!]^{\mathit{ParProc}} P =$
$$[\![\, (x_1 : T_1; \ldots; x_n : T_n \bullet \mathit{Proc}) \\ [c : \mathit{usedC}(\mathit{Proc}) \bullet c\_x\_1 \ldots x\_n.x_1. \ \ldots \ .x_n] \,]\!]^{\mathit{ParProc}} P$$

The process $P[c : \mathit{used}(\mathit{Proc}) \bullet c\_x\_1 \ldots x\_n.x_1. \ \ldots \ .x_n]$ is that obtained from P by changing all the references to a used channel c by a reference to the channel $c\_x\_1 \ldots x\_n$, with synchronisation $x_1. \ \ldots \ .x_n$.

An instantiation of an indexed process is translated as an invocation of a parametrised process. However, the same syntactic substitution as the one present in the rule above is made before the translation.

Rule 6.30 $[\![\, (x_1 : T_1; \, \ldots; \, x_n : T_n \odot Proc) \lfloor v_1, \ldots, v_n \rfloor \,]\!]^{Proc} =$
$$[\![\, ((x_1 : T_1; \, \ldots; \, x_n : T_n \bullet Proc) \\ [c : usedC(Proc) \bullet c\_x\_1 \ldots x\_n.x_1. \ \ldots \ .x_n]) \, (v_1, \ldots, v_n) \,]\!]^{Proc}$$

Finally, if the instantiation uses the process name, we may translate it as follows.

Rule 6.31 $[\![\, N \lfloor v_1, \ldots, v_n \rfloor \,]\!]^{Proc} = [\![\, N(v_1, \ldots, v_n) \,]\!]^{Proc}$

This concludes the translation of indexed processes; in the next sections we extend our translation strategy further by allowing generic and multi-synchronised channels.

6.2.9 Generic Channels

In this section, we deal with generic channels. We consider the declaration of the channel *lamp* used in our case study, which is used throughout this section to illustrate the definitions.

$$\textbf{channel}[T] \; lamp : T \times OnOff$$

This declaration introduces a family of channels *lamp*. In this declaration, T is used as a parameter used to determine the type of the values that are used in the communication of a value of type $OnOff$.

Each generic typing variable in a generic channel declaration is implemented as an additional dimension in an array of channels: each element represents a possible instance of the channel. By way of illustration, we have the instantiation of the generic channel $lamp[AreaId]$ from our example, which is implemented as $lamp[Type.AREA\_ID]$.

The simplification we did for synchronisation channels still holds throughout this section: we consider communications of the following form.

$$\mathsf{Comm} \quad ::= \quad \mathsf{N} \, [\mathsf{Exp}^{+}] \, .\mathsf{Exp}^{*} \, \mathsf{CParameter} \mid \mathsf{N} \, .\mathsf{Exp}^{*} \, \mathsf{CParameter}$$

The only difference from the Section 6.2.7 is the possibility of generic channel instantiations.

Our translation strategy assumes that every type used within the system is already implemented in Java. Besides, the class `Type` has an integer constant for each type used within the system. The translation strategy translates references to a type into a reference to the corresponding constant in class `Type`.

A generic process declaration is translated as a process parametrised by the types used in the declaration. For this reason, if we have a generic process P, we consider the type arguments as arguments of P in its translation, and replace every reference to that type identifier, by the primitive value of the integer given as argument. Besides, any reference to the generic type variable is replaced by a reference to the superclass `Type`. The typing variables are not defined as types; we assume that the function *JType* returns the names given as arguments in these cases.

Rule 6.32 $[\![\,\textbf{process}\ P[T_0, \ldots, T_n] \mathrel{\widehat=} \mathit{Decl} \bullet \mathit{Proc}\,]\!]^{\mathit{Proc}}\ \mathit{proj} =$

$$([\![\,\textbf{process}\ P \mathrel{\widehat=} t_0 : \mathbb{Z};\ \ldots;\ t_n : \mathbb{Z};\ \mathit{Decl} \bullet \mathit{Proc}]\!]^{\mathit{Proc}}\ \mathit{proj}) \left[\begin{array}{lcl} \texttt{Type}, \ldots, \texttt{Type}, & & \mathit{JType}\ T_0, \ldots, \mathit{JType}\ T_n, \\ \texttt{t\_0.intValue(),} & / & \texttt{Type.}(\mathit{CJType}(T_0))\texttt{,} \\ \ldots, & & \ldots \\ \texttt{t\_n.intValue()} & & \texttt{, Type.}(\mathit{CJType}(T_n))\texttt{]} \end{array}\right]$$

For a given type T, the function *CJType*, returns the name of the Java type of T with all the letters capitalised.

Instead of using the previously defined function *ArrayDimSync*, the functions *VisCDecl*, *HidCDecl*, *VisCArgs*, and the instantiation of hidden channels, use the function *ArrayDim* defined below in order to find out the dimension of the possible array of channels.

The function *ArrayDim* receives three arguments: a list *genPars* of the generic parameters of the channel, and the two arguments used by the function *ArrayDimSync*.

Definition 6.4

$$\begin{array}{l} \mathit{ArrayDim}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc} = \\ \quad \textbf{let}\ \ \mathit{dim} = \#\mathit{genPars} + (\mathit{ArrayDimSync}\ \mathit{types}\ \mathit{sc})\ \textbf{in}\ \texttt{[]}^{\,\mathit{dim}} \end{array}$$

This function adds the number of generic parameters used in the declaration of a given channel to the result of *ArrayDimSync*.

For channel *lamp*, we have that $\mathit{genPars} = [T]$, $\mathit{types} = [T, \mathit{OnOff}]$, and $\mathit{sc} = C$. The application of function *ArrayDim* with these arguments returns the string `[][]`. For this reason, the function *VisCDecl* returns the Java code `private ChannelOutput[][] lamp`. The array dimension for this channel is two, one for being a single generic channel and another for being a synchronisation channel.

As in Section 6.2.7, the changes in the translation strategy are concerned with the declaration, instantiation, and use of the generic channels. Regarding the channel declaration and instantiation, the only difference from Section 6.2.7 is that we replace the use of the function *InstArraySync* by the use of the function *InstArray*, which also instantiates an array of channels. First, it deals with the dimensions related with the generic parameters, and then it uses the previously defined function *InstArraySync*, in order to deal with dimensions that are originated from synchronisation. It receives the generic parameters (*genPars*) and the types (*types*) used in the declaration of the channel, a value *sc* of type *SC* indicating if the channel is a synchronisation or a communication channel, and a list of all types used within the system (τ).

For each generic parameter, this function instantiates an array of channels with a dimension determined by the function *ArrayDim*. This instantiation uses an auxiliary function *GenericInst*, which declares an element in the array for each type T_n used within the system. Finally, when all the generic parameters have been dealt with, the function invokes the function *InstArraySync*, in order to instantiate any further arrays related to possible synchronisation.

Definition 6.5

$$
\begin{array}{l}
\mathit{InstArray}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc}\ \mathit{tEnv} = \\
\quad \mathbf{let}\ \mathit{dim} = (\mathit{ArrayDim}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc}) \\
\quad \mathbf{in\ if}\ (\#\mathit{genPars} > 0)\ \mathbf{then} \\
\qquad\quad \mathtt{new\ Any2OneChannel}\ \mathit{dim}\{ \\
\qquad\qquad \mathit{GenericInst}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc}\ \mathit{tEnv}\ \mathit{tEnv}\ \mathtt{\}} \\
\qquad \mathbf{else}\ \mathit{InstArraySync}\ \mathit{types}\ \mathit{sc}
\end{array}
$$

For each type T within the system, the function *GenericInst* invokes the function *InstArray* in order to deal with the remaining (if any) generic parameters. However, each element corresponds to an instantiation of the first generic variable in the list with a certain type T; for this reason, we change the channel *types* by using the function *replace* in order to replace every reference to the the first generic parameter in the list (*head genPars*) by the type T.

Definition 6.6

$$
\begin{array}{l}
\mathit{GenericInst}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc}\ \mathit{tEnv}\ [T] = \\
\quad \mathit{InstArray}\ (\mathit{tail}\,\mathit{genPars})\ (\mathit{replace}(\mathit{head}\,\mathit{genPars}, T, \mathit{types}))\ \mathit{sc}\ \mathit{tEnv} \\
\mathit{GenericInst}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc}\ \mathit{tEnv}\ T :: \mathit{TS} = \\
\quad \mathit{InstArray}\ (\mathit{tail}\,\mathit{genPars})\ (\mathit{replace}(\mathit{head}\,\mathit{genPars}, T, \mathit{types}))\ \mathit{sc}\ \mathit{tEnv}\,\mathtt{,} \\
\quad \mathit{GenericInst}\ \mathit{genPars}\ \mathit{types}\ \mathit{sc}\ \mathit{tEnv}\ \mathit{TS}
\end{array}
$$

Our example falls in the first case of function *InstArray*. As previously discussed, for this channel, we have $genPars = [T]$, and so $\#[T] = 1 > 0$. As we already know, the array dimension for this channel is two, so we have the following instantiation for channel *lamp*.

```
this.switchLamp = new Any2OneChannel[][]{
                              GenericInst [T] [T, OnOff] C tEnv tEnv };
```

In the specification of our case study, eight types were used in total, and hence, the application of function *GenericInst* results in the following comma-separated list of invocation of the function *InstArray*.

$$
\begin{array}{l}
\mathit{InstArray}\ (\mathit{tail}\,[T])\ [\mathit{AreaId}, \mathit{OnOff}]\ C\ \mathit{tEnv}\,\mathtt{,} \\
\ldots\ \mathtt{,} \\
\mathit{InstArray}\ (\mathit{tail}\,[T])\ [\mathit{AlarmStage}, \mathit{OnOff}]\ C\ \mathit{tEnv}
\end{array}
$$

Each of the lines corresponds to a certain type within the system. Notice that the function *replace* has replaced the parameter T, in the *types* list $[T, OnOff]$ by each corresponding type. For instance, in the first line, which corresponds to the type *AreaId*, we have that the *types* list given to the function *InstArray* as argument is $[AreaId, OnOff]$. Besides, we have that $tail\,[T] = [\,]$. For this reason, for each of the eight elements above, we have that the function applications above return the following results.

$InstArraySync\ [AreaId, OnOff]\ C$
...
$InstArraySync\ [AlarmStage, OnOff]\ C$

For example, the result for the first of them is presented below. By applying the definition of *InstArraySync*, we see that the dimension of the array is now one. Hence, we have the following result.

`new Any2OneChannel[]{` $TypeInstSync\,[AreaId, OnOff]\ C\ 2$`}`

The function *TypeInstSync* creates two comma-separated invocations to the function *InstArraySync* as follows.

$InstArraySync\,[OnOff]\ C$ `,` $InstArraySync\,[OnOff]\ C$

Again, if we follow the definition of *InstArraySync*, we get to the base case, which, since we have a communication channel (C), returns a single channel instantiation.

```
new Any2OneChannel(),new Any2OneChannel()
```

This concludes the instantiation of the channel *lamp*. In Figure 6.6, we present the whole Java code for the instantiation of this channel. We are left now with the usage of these channels.

As for synchronisation channels, just a few rules must be redefined. These redefinitions are straightforward. Basically, we extend the definitions for synchronisation channels, by taking into account possible generic channel instantiations. For a given input or output channel c, the type of the communicated value (*commType*) is given by the Java type of the last element in the list (*last*) of types of c. This is the type used to declare the new variable.

Rule 6.33 $[\![c\ [T_0, \ldots, T_n].e_0\ \ldots\ .e_m?x \rightarrow Act]\!]^{Act} =$
  let $commType = JType(last\ (snd\,(\delta\ c)))$
  in `{` $commType$ `x = (`$commType$`)c[Type.(`$CJType\ T_0$`)]` ... `[Type.(`$CJType\ T_n$`)]` `[`$JExp\ e_0$`]` ... `[`$JExp\ e_n$`].read();`
    $[\![Act]\!]^{Act}$`}`

An output still writes to the channel. However, as we have arrays of channels, we must guarantee we access the correct element in the array.

Rule 6.34 $[\![c\ [T_0, \ldots, T_n].e_0\ \ldots\ .e_m!x \rightarrow Act]\!]^{Act} =$
  `c[Type.(`$CJType\ T_0$`)]` ... `[Type.(`$CJType\ T_n$`)]`
  `[`$JExp\ e_0$`]` ... `[`$JExp\ e_n$`].write(`$JExp\ x$`);`
  $[\![Act]\!]^{Act}$

Finally, synchronisation channels simply read from an channel or write a null value to

```
this.switchLamp =
    new Any2OneChannel[][]{
        // Type AreaId
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel()},
        // Type ZoneId
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel()},
        // Type LampId
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel() },
        // Type FaultId
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel(), new Any2OneChannel()},
        // Type OnOff
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel()},
        // Type Mode
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel() },
        // Type SwitchMode
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel()},
        // Type AlarmStage
        new Any2OneChannel[]{ new Any2OneChannel(), new Any2OneChannel(),
                              new Any2OneChannel() } };
```

Figure 6.6: Instantiation of channel *lamp*

the corresponding channel.

Rule 6.35 $[\![c\ [T_0, \ldots, T_n].e_0\ \ldots\ .e_m \rightarrow Act]\!]^{Act} =$
`c[Type.`$($*CJType* $T_0)$`]`$\ldots$`[Type.`$($*CJType* $T_n)$`]`
`[`*JExp* e_0`]`$\ldots$`[`*JExp* e_n`].read();`
$[\![Act]\!]^{Act}$

provided $\nu\ c \in \{ I, A \} \vee \iota\ c \in \{ I, A \}$ ❑

Rule 6.36 $[\![c\ [T_0, \ldots, T_n].e_0\ \ldots\ .e_m \rightarrow Act]\!]^{Act} =$
`c[Type.`$($*CJType* $T_0)$`]`$\ldots$`[Type.`$($*CJType* $T_n)$`]`
`[`*JExp* e_0`]`$\ldots$`[`*JExp* e_n`].write(null);`
$[\![Act]\!]^{Act}$

provided $\nu\ c = O \vee \iota\ c = O$ ❑

As for synchronisation channels, in the case of external choice, we must redefine some of the auxiliary functions to take into account the existence of arrays of channels; each channel in an array of channels is considered as a different visible channel. These redefinitions are pretty straightforward. For instance, the function *ICAtt*, which is used in the declaration of the array of channels that take part in the external choice, takes into account possible generic channels and synchronisation values in the channels. This means that channels of the form $c.[T_0, \ldots, T_n]x_0 \ldots .x_m$ are translated to the communication presented below.

`c[Type.`$($*CJType* $T_0)$`]`$\ldots$`[Type.`$($*CJType* $T_n)$`][`*JExp* x_0`]`$\ldots$`[`*JExp* x_m`]`

The function *DeclCs* is used to declare one constant for each channel that takes part in the external choice. It returns a semicolon-separated list of `int` constant declarations, one for each channel in the given channel list. Its redefinition, however, takes into account the possible existence of arrays of channels. Therefore, for each channel c in the form presented above, we have a new constant declaration as follows.

`final int CONST_C_`$($*CJType*$(T_0))$`_`$\ldots$`_`$($*CJType*$(T_n))$`_X_0_`$\ldots$`X_m = n;`

Finally, the redefined function *Cases*, which returns a sequence of Java `case` blocks, one for each initial channel in given channel list, returns a different case block for each element in an array of channels that take part in the external choice.

As discussed before, the types are declared as arguments of a generic process. For this reason, we must use the constants which represent each of the types used in the instantiation. These are given as Java `Integer`s, which are constructed using the corresponding constants, to the constructor of the class corresponding to the process that is being instantiated.

Rule 6.37 $[\![\, N[T_0,\ldots,T_n](e_0,\ldots e_n)\,]\!]^{Proc} =$

```
(new CSProcess(){
    public void run() {
        (new N(new Integer(Type.(CJType T0)),...,
               new Integer(Type.(CJType Tn)),
               (JExp e0),...(JExp en),
               ExtChans ν)).run();
    }
}).run();
```

Besides, for parametrised processes, we have that *JExp e* is also given to the constructor, for each parameter e, used to instantiate the process. Finally, as expected, a comma-separated list of visible channels of the process is also used to instantiate the process.

6.2.10 Multi-synchronisation

In this section, we deal with multi-synchronisation channels. First, we present some Java components that were implemented by us for use in this translation. Then, we present the translation rules.

We implement multi-synchronisation using a centralised solution based on the work presented in [101]: the distribution of a multi-synchronisation is replaced by a process that controls the multi-synchronisation in a given channel, and by client processes that potentially synchronise on the channel. Four components were implemented by us using JCSP and are used here: the first two are the process that represents the multi-synchronisation controller (`MultiSyncControl`) and the process that represents the multi-synchronisation client (`MultiSyncClient`). Their implementation follows directly from our extension of the protocol presented in [101]. The controllers are implemented using a infinite loop. For this reason, their direct use in the implementation of the process will never reach termination; we use two other components that guarantee that, when the process terminates, the controllers also terminate. All the controllers within a process are managed by a process `ControllersManager` and the process itself is managed by the process `ProcessManagerMultiSync`, which uses the channel `endManager` to signal to the `ControllersManager` that the process has terminated. This leads to the controllers manager stopping the execution of every controller that it is responsible for.

For each channel involved in a multi-synchronisation, we have a controller; each time a process is willing to engage in a multi-synchronisation, we must instantiate a new client process and run it. At the end of the execution, possibly communicated values can be retrieved from the client.

In Figure 6.7, we illustrate an architecture using these components for two channels involved in a multi-synchronisation and a process whose behaviour is a parallel composition of four individual processes. In this example, we have one `MultiSyncControl` for each channel, and each individual process instantiates its own `MultiSyncClient`. The controllers use an array of channels `fromSync` to communicate with each of their clients.

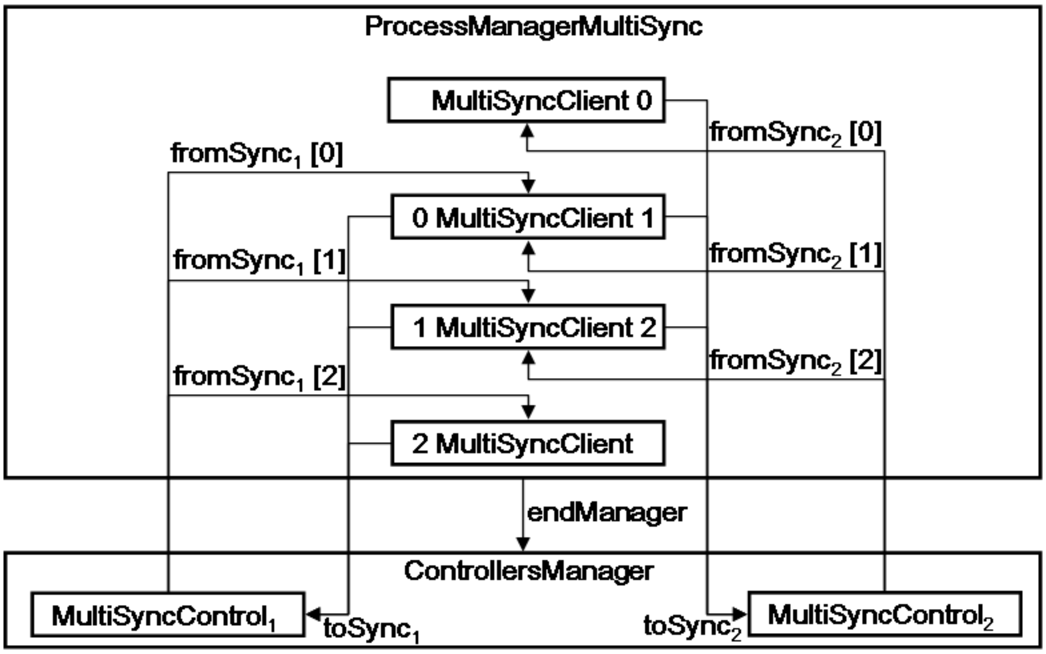


Figure 6.7: Architecture for the Multi-synchronisation components

The clients share a channel `toSync` to communicate with their controller. This channel is not multi-synchronised, since in JCSP communications happen only between two processes. The three top-most clients synchronise on the channel that is controlled by the right-hand side controller, and the three bottom clients synchronise in the channel controlled by the left-hand side controller. Each of the clients have a different identification regarding each of the controllers. For instance, the second client from the top is identified as client zero on the left, and as client one on the right. We extend the work presented in [101]: clients may take part in more than one multi-synchronisation, in non-multi-synchronised communications, and values may be carried through channels involved in a multi-synchronisation. All the controllers are managed by the `ControllersManager` and the process itself is managed by the `ProcessManagerMultiSync`; they communicate via the channel `endManager`.

Another environment is considered to be available throughout the translation strategy: the environment $\omega : \mathsf{N} \nrightarrow ((\mathsf{N} \rightarrowtail\!\!\!\!\!\nrightarrow \mathbb{Z}) \times \mathbb{Z} \times \mathsf{N})$ includes a triple for every name of channel involved in a multi-synchronisation. The first element is a function that gives an identification number for every process involved in the multi-synchronisation. Our strategy considers that these identifications start from zero, and are incremented by one for each process. For instance, consider a channel c, in which processes P_0, P_1, and P_2 synchronise. A possible identification function would be $\{P_0 \mapsto 0, P_1 \mapsto 1, P_2 \mapsto 2\}$. The second element in the triple is the number of processes that are involved in the multi-synchronisation. This can be easily calculated from the cardinality of the domain of the identification function, but we keep it for conciseness in the definitions. The third element in the triple is the name of the process that is writing to the channel; following a limitation from JCSP, all other processes are considered readers.

Consider a system with only one channel c used for multi-synchronisation and a process P whose execution is the parallel composition of three other processes P_0, P_1, and P_2 (the writer) that synchronise on c. In this case, the environment ω would be $\{c \mapsto (\{P_0 \mapsto 0, P_1 \mapsto 1, P_2 \mapsto 2\}, 3, P_2)\}$.

In order to create a `MultiSyncControl` process that controls the synchronisation of channel c, the user must give the array of channels `from_c`, and the channel `to_c` that the clients use to communicate with the controller, as arguments. The number of clients can be easily retrieved from the ω environment ($JExp(snd(\omega\ c))$).

```
Any2OneChannel[] from_c = Any2OneChannel.create(3);
Any2OneChannel to_c = new Any2OneChannel();
MultiSyncControl c = new MultiSyncControl(from_c, to_c);
```

The constructor of a controller manager takes its controllers as an argument; the other argument that it takes is the channels used to communicate with the process manager, which must be instantiated.

```
Any2OneChannel endManager = new Any2OneChannel();
ControllersManager cManager = new ControllersManager(endManager,c);
```

The process manager receives the communication channel `endManager` and the process P it must manage. For conciseness, we omit the arguments needed to instantiate the process P.

```
CSProcess p = new P(...);
ProcessManagerMultiSync pManager =
    new ProcessManagerMultiSync(endManager,p);
```

Finally, the execution of the process P is actually implemented as the parallel composition of the process manager and the controllers manager as presented below.

```
(new Parallel(new CSProcess[]{cManager,pManager})).run();
```

The instantiation of the clients requires a little bit more of information. For each multi-synchronisation c on which the process takes part we must create a synchronisation object that contains the channel used by this client in the array of channels `from_c`, the channel `to_c`, the identification of the process in this multi-synchronisation (0), and the identification of the writer in that channel (2). For instance, in the translation of process P_0, we create the following synchronisation object for channel c.

```
Object[] sync = new Object[]{from_c[0], to_c, 0, 2};
```

Next, we create a `Vector` that contains all these multi-synchronisation objects.

```
Vector sqOfSyn = new Vector(); sqOfSyn.addElement(sync);...
```

A `Vector` of channels that are not involved in a multi-synchronisation is also used in the instantiation of a client.

For instance, consider we have a channel `nm`, which is not multi-synchronised. We have the following Java code.

```
Vector sqOfNSyn = new Vector(); sqOfSyn.addElement(nm);
```

Finally, we instantiate the client and execute it as follows.

```
MultiSyncClient client = new MultiSyncClient(sqOfSyn,sqOfNSyn,v);
client.run();
```

The last argument `v` is the value communicated through the channel. If this client is the writer, this is the value that will be communicated to the readers once the synchronisation happens. If this process is not the writer, we use `null` instead of `v`.

Most of the extension for dealing with multi-synchronisation is done simply by replacing code in the Java code generated by the translation strategy presented so far. These replacements change only those parts of the Java code that are related to the multi-synchronisation channels. Basically, every reference to a channel involved in a multi-synchronisation is replaced by a reference to the channels used in the communication with the multi-synchronisation controller.

Furthermore, the translation of external choice is changed in order to deal with channels involved in a multi-synchronisation, and the translation of the body of a process is also changed in order to include the execution of the controllers for the channels involved in a multi-synchronisation that are hidden within that process. All the changes are discussed later in this section.

The first code substitution deals with the declaration of every hidden channel c involved in a multi-synchronisation within a process. We replace every declaration of such channel by the declaration of a channel `private Any2OneChannel to_c` and the channel array `private Any2OneChannel[] from_c` used to communicate with the multi-synchronisation controller.

In a similar way, we also make a substitution of the declarations of visible channels involved in a multi-synchronisation. However, since the control of reading and writing in a channel involved in a multi-synchronisation is now left to the synchronisation controller, we do not make any distinction between input and output channels. The same applies in the declaration of the class constructors arguments that are related to the channels involved in a multi-synchronisation.

Next, we replace instantiations of these channels by instantiations of the channels that make the communications between servers and clients as follows. The environment ω gives how many processes take part in the multi-synchronisation.

```
this.from_c = Any2OneChannel.create(JExp(fst(ω c)));
this.to_c = new Any2OneChannel();
```

The initialisation of visible channels in the constructor is also replaced by the assignment of both channels related to c: `this.from_c = newFrom_c; this.to_c = newTo_c;`. We consider *MSyncDCSubst* as the composition of all these substitutions; this function will be used later in this section.

Every access to a channel involved in a multi-synchronisation must be made in the ways explained before: each time that a multi-synchronisation is used, we must instantiate a client and execute it. Given an index *ind*, the function *InstMultiSync* returns the Java code that implements a multi-synchronisation on channel c within a process P. The index of the right channel within the array `from_c` is the result of the expression $JExp((fst(\omega\ c))\ P)$, which is the Java expression corresponding to the identity of the process in the synchronisation. The identity of the writer is given as the application of the function that gives us the processes' identities to the name of the writer process, thus $JExp((fst(\omega\ c))\ (trd(\omega\ c)))$.

We present below how a multi-synchronisation on c within P_0 must be implemented.

```
Vector sqOfSyn_0 = new Vector();
Object[] sync_0 = new Object[]{from_c[0],to_c,0,2};
sqOfSyn.addElement(sync_0);
MultiSyncClient client_0 =
   new MultiSyncClient(sqOfSyn_0, new Vector(),null);
client_0.run();
```

We replace any channel reading that stores the read value and then, we replace the remaining communications. The reading $T\ x$`=(`T`)c.read()` from every multi-synchronised channel c, where T is any java type, is replaced by the following: we execute a client, as the one presented above, and finally, we retrieve the communicated value from the client $T\ x$`=(`T`)client_`*ind*`.getValueTrans();`. Writing (`c.write(x)`) to a channel and reading (`c.read()`) from a channel (if we do not store the read value) are also replaced by a client execution. Processes that write a value `x` are considered to be the channel writer; therefore, in this case, we also replace the `null` value used in the instantiation of the client, by `x`. The combination of both substitutions is called *ChanUseSubst*, and replaces any use of channels involved in multi-synchronisation.

The only rules that need changes are those for parametrised processes and for external choice. The new translation of external choice is similar to the one presented before. The only change from Rule 6.10, is that it does not use the `Alternative` JCSP class, but a multi-synchronisation client. For every multi-synchronised channel c involved in the external choice, we create a synchronisation object corresponding to c and insert it the vector `sqOfSync_`*ind*; the remaining channels are inserted in the vector `sqOfNSync_`*ind*. Finally, as explained before, we create a multi-synchronisation client `client_`*ind* and execute it. At the end of its execution, `client_`*ind* has the index of the chosen channel, and possibly a communicated value. Since there is no output guards, every channel that takes part in an external choice is not willing to write anything to the channel, hence, the communicated value used to instantiate the *MultiSyncClient* is `null`. Finally, the translation of `switch` block remains almost the same: the choice is actually retrieved from the client using the method `getChosen`, which returns the index of the chosen channel.

In the previous translation of external choice, after choosing a channel in the `switch` block, the first line of the code is the translation of the channel reading. However, in the case of multi-synchronisation, this is already done by the multi-synchronisation client. For this reason, the translation of the `case` bodies related to the channels involved in a multi-synchronisation are slightly changed: we replace `(`T`)`x `= (`T`)c.read()` by `(`T`)`x `=(`T`)client_`ind`.getValueTrans()` and remove the remaining readings.

Every channel is instantiated within the process that hides it; otherwise it is received and initialised in the constructor. For this reason, we chose to instantiate the multi-synchronisation controller for a given channel in the same class in which c is instantiated. First, we need to retrieve the information of which multi-synchronisation channels are hidden within a process. This can be easily defined as the intersection of the hidden channels of a process (dom ι) and the multi-synchronisation channels (dom ω). We emphasise that the environment ι stores information for every hidden channel in a certain process.

$$\mathit{MultiSyncChs} \mathrel{\widehat=} \operatorname{dom}\iota \cap \operatorname{dom}\omega$$

For a given process whose $\mathit{MultiSyncChs}$ is $\{c_1, \ldots, c_n\}$, we have that the process that represents the parallel composition of a controller for every channel in $\mathit{MultiSyncChs}$ is defined as $\mathit{MultiSyncControl}(\mathit{from\_}c_1, \mathit{to\_}c_1) \parallel \ldots \parallel \mathit{MultiSyncControl}(\mathit{from\_}c_n, \mathit{to\_}c_n)$. We redefine Rule 6.2 for parametrised processes, by replacing the declarations, constructor arguments, and initialisation of the multi-synchronised channels accordingly. Furthermore, the `run` method body has two changes: the references to the multi-synchronised channels are also changed accordingly and it also deals with the instantiation of the multi-synchronisation controllers. In the declaration part and in the constructor, we replace the references to multi-synchronised channels using the substitution $\mathit{MSyncDCSubst}$; in the body of the `run` method we replace references to the multi-synchronised channels using the substitution $\mathit{ChanUseSubst}$.

Rule 6.38 $[\![D \bullet P]\!]^{\mathit{ParProc}} N \mathrel{\widehat=}$

$$\mathit{Subst}\ \mathit{MSyncDCSubst}\ (\operatorname{dom}\omega) \left(\begin{array}{l} (\mathit{ParDecl}\ D)\ (\mathit{VisCDecl}\ \nu)\ (\mathit{HidCDecl}\ \iota) \\ \texttt{public}\ N\texttt{(}\mathit{ParArgs}\ D\texttt{,}\mathit{VisCArgs}\ \nu\texttt{)}\ \texttt{\{} \\ \quad (\mathit{MAss}\ (\mathit{ParDecl}\ D)\ (\mathit{ParArgs}\ D)) \\ \quad (\mathit{MAss}\ (\mathit{VisCDecl}\ \nu)\ (\mathit{VisCArgs}\ \nu)) \\ \quad \mathit{HidCC}\ \iota\ \texttt{\}} \end{array}\right)$$

```
public void run(){
    final Any2OneChannel endManager = new Any2OneChannel();
```

$$\begin{array}{l} \mathit{Subst}\ \mathit{ChanUseSubst}\ (\operatorname{dom}\omega) \\ \quad [\![\mathit{ProcessManagerMultiSync}(P) \\ \quad \parallel \mathit{ControllersManager} \left(\begin{array}{l} \mathit{MultiSyncControl}(\mathit{from\_}c_1, \mathit{to\_}c_1) \\ \parallel \ldots \\ \parallel \mathit{MultiSyncControl}(\mathit{from\_}c_n, \mathit{to\_}c_n) \end{array}\right)]\!]^{\mathit{Proc}} \end{array}$$

```
}
```

provided $\mathit{MultiSyncChs} \neq \varnothing$ ❑

The application $\mathit{Subst}\ s\ cs\ jc$, where jc is a Java code, returns the java code resulting from applying the substitution s to jc, for every channel c in the set of channels cs.

One last rule that must be changed is the rule that translates process invocations. This is needed to deal with the fact that we actually have invocations of three very special processes: *MultiSyncControl*, *ProcessManagerMultiSync*, and *ControllersManager*. These processes need particular translations: for instance, we do not need to give each of the external channels to these processes.

In the translation of the first one, the *MultiSyncControl*, we use only two arguments: the array of channels used by the controller to communicate with the clients, and the channel used to communicate with the controller.

Rule 6.39 $[\![\ MultiSyncControl(from\_c, to\_c)\ ]\!]^{Proc} =$

```
(new MultiSyncControl(from_c,to_c)).run();
```

The translation of the *ProcessManagerMultiSync* uses only the process which is going to be controlled by this component.

Rule 6.40 $[\![\ ProcessManagerMultiSync(Proc)\ ]\!]^{Proc} =$

```
(new ProcessManagerMultiSync(
    endManager, new CSProcess(){ public void run(){ [[Proc]]^Proc }}
)).run();
```

Finally, the translation of *ControllersManager* is very similar to the one presented above for *ProcessManagerMultiSync*; the only change is that it instantiates and executes an object of class `ControllersManager`, using the same arguments.

This extension of the translation strategy presented in [73] was vital in the implementation of our case study since multi-synchronisation plays a major role in our system.

6.3 Implementing the Fire Control System

The implementation of the whole system can be found in [71]. After translation, the classes that implement the processes are located in the package `processes`. Figure 6.8 presents a UML class diagram of this package after the translation strategy was applied to our case study. We highlight the core of the system which was presented in this paper, the process `ConcreteFireControl`. This process hides multi-synchronisation channels, and for this reason, it is the one responsible for instantiating the multi-synchronisation controllers, and their respective managers, for each of these channels. On the other hand, the process that implements each of the areas in the fire control system (`Area`), the process that implements the display controller (`DisplayController`), and the process that implements the core of the fire control (`FireControl`) take part in multi-synchronisation, and hence, instantiate multi-synchronisation clients.

In order to run the whole system, we have created a parallel composition of the `ConcreteFireControl` with a process that represents a `Clock`. The external devices where also implemented. A `Keyboard` may be used to input signals to the system. The `Output` process encapsulates the `Alarm` and the `Display`. The last is composed of a buzzer and the lamps. There are three different instantiations of the lamps: the `FireLamps` indicate where a fire has been detected; the `GasReleasedLamps` indicate where gas has been

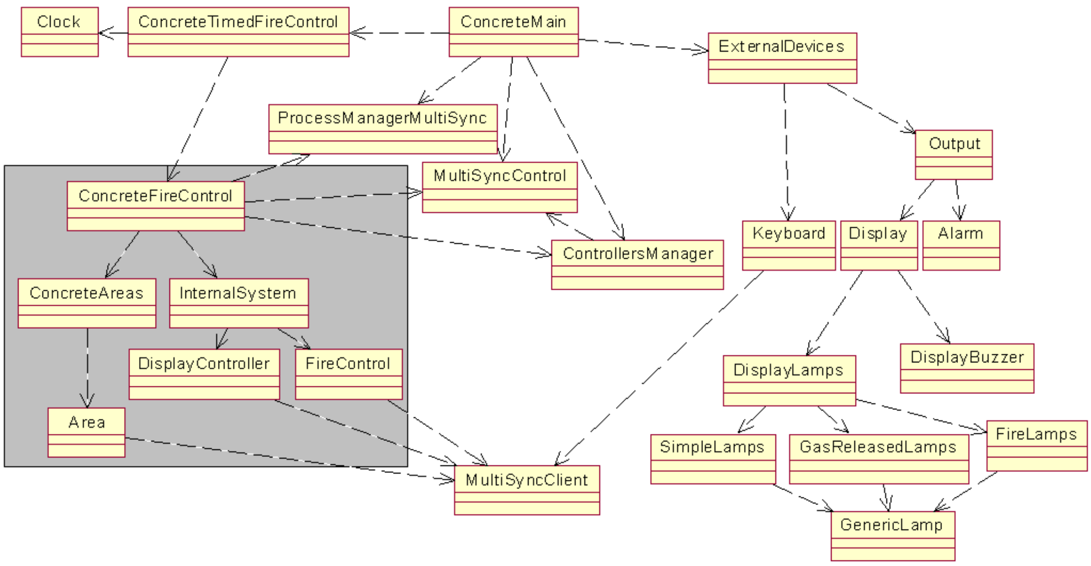


Figure 6.8: Fire Control System Class Diagram (processes only)

discharged; and the remaining lamps are implemented within the process `SimpleLamps`. All the lamps are instantiations of the generic process `GenericLamp`. The parallel composition of the `ConcreteTimedFireControl` with the `ExternalDevices` represents the whole system, `ConcreteMain`. The implementation has also included typing classes, utility classes, which are also part of our translation strategy (*e.g.,* `RandomGenerator`), and graphic interface classes, which, although not arising from the translation, where implemented in order to allow us to interact with the system.

In Figure 6.9, we present a snapshot of the execution of the process `ConcreteMain`. This interface contains the following elements: gas lamps for areas 0 and 1, fire lamps for zones 0 to 5, one fault lamp for each possible fault within the system, the lamp that indicates whether the system is on or off, a clock that shows if the clock is counting down, a keyboard that can be used by the user to simulate inputs to the system, and an alarm in the form of a progress bar: an empty bar indicates that the alarm is off, a half filled bar indicates a first stage alarm, and a full filled bar indicates a second stage alarm. Besides, a sound is also played if the display buzzer is switched on.

In this snapshot, we have that fire has been detected in zones 0, 1, and 2, and that three faults have been detected: second line fault, power fault, and isolate remote signal. In this example, the system is running in automatic mode. As specified, the fact that fire has been detected in two zones in the same area, has started the counting down of the clock and has set the alarm to its second stage. After the conclusion of the counting down, the gas is discharged in area 0 and this is indicated in the display by switching the gas lamp 0 on.

The implementation of the fire control system using the translation strategy yielded 5400 lines of Java code [71]. Unfortunately, no access to the original source code was given; this would allow us to compare the size and complexity of the source codes. Throughout

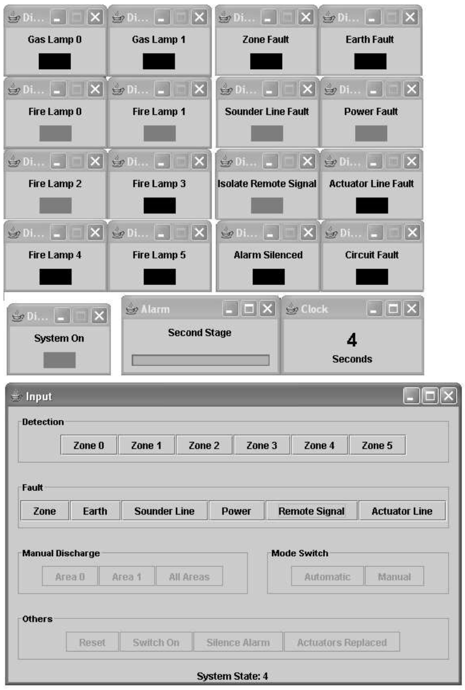


Figure 6.9: Fire Control System Graphic Interface

the translation of the case study, we could verify the correctness of our translation strategy, and even simplify the definitions of some rules. Furthermore, the translation of the case study also provided us with a industrial example of application of the strategy.

6.4 Final Considerations

The translation strategy presented in this work extends the one we presented in [73], by including synchronisation and generic channels, indexing operators, generic processes, and multi-synchronisation. The strategy has been used to implement several programs, including a quite complex fire control system developed from its abstract centralised specification [71], which is also presented here. The application of the translation rules was straightforward; only human errors, which could be avoided using the prototype of the translation tool [44] that implements the translation strategy presented in this chapter, raised problems. The choice of JCSP was motivated by the extent support of the JCSP implementors. Furthermore, the direct correspondence between many CSP and *Circus* constructs is a motivation for extending JCSP to support *Circus*, instead of creating another library from scratch.

The strategy presented in this chapter is slightly different from the one we present in [76]; here, we present the corrections to the problems pointed out by the work done by Freitas [44]. In [76], we did not have the managers in the implementation of multi-synchronised channels; this led to a non-terminating behaviour of the controllers. In this chapter, we introduced the managers in order to stop the execution of the controllers at the end of the process execution.

Another correction done to the strategy presented in [73, 76] regards the copies of the state components and local variables in the parallel composition of actions. Previously, we only had copies for the elements in the partitions used in the declaration of the parallel composition. Both sides of the composition should have copies of all the state components and local variables in scope. In Section 6.2, we correct this by creating these copies in both sides of the parallel composition.

In this chapter, we have considered that the types used in the system are already implemented in Java. In [44], Freitas presents how this can be achieved for free types, abbreviations, and integers. Another extension to our strategy presented in [44] is in the translation of the parallel composition of actions: Freitas allows actions in a parallel composition to invoke other actions. Furthermore, the translation of nested external choices and guards is also presented in [44]. Finally, in Section 6.2.10, for simplicity, the translation of multi-synchronised channels is defined in terms of some substitutions in the Java code. The design used in [44] allows the direct translation of multi-synchronised generic channels without the need of any substitutions.

Certainly, code generated by hand could be simpler. For instance, the translation of compound processes do not always need anonymous inner classes; they are used in the rules for generalisation purposes. However, our experiments have shown no significant improvement in performance after simplification.

Throughout the translation we assume that the specification has been refined into a specification that meets the translation strategy's requirements. For instance, all operation schemas and specification constructs have already been refined. Another requirement is the order of the paragraphs: we assume that, in the *Circus* program to be implemented, we have first Z paragraphs, then, channel declarations, and finally, process declarations. This, however, can be achieved with a simple reordering of the paragraphs. The next requirement concerns the Z paragraphs used to group channel declarations, and channel sets. Our strategy requires they have already been expanded. This can also be achieved with a simple refinement. The only Z paragraphs considered are axiomatic definitions of the form $v : T \mid v = e$, free types, or abbreviations. The considerations of other types of paragraphs is left as future work.

Due to JCSP limitations, we consider a restricted set of communications: untyped inputs, outputs, synchronisations, generic channels, synchronisations $c.e$ over a channel c with expression e, and multi-synchronisations. Strategies to refine out the remaining forms of communication and output guards are left as future work.

JCSP itself restricts our strategy in the translation of parallel operator. It does not support the definition of a synchronisation channel set: the intersection of the alphabets determines the synchronisation channels set.

Not all iterated operators are treated directly. The translation of iterated parallel operator and interleaving of actions requires their expansion. For external choice, expansion is required for both the action and the process operator, due to the need to determine their initials.

An important piece of future work is the implementation of a tool to support the translation strategy; a prototype of such a tool, which is based on the translation strategy presented in this thesis can be found in [44]. In order to prove the soundness of such a tool, the proof of the translation rules presented here would be necessary. In [44], the first step towards this proof is given: Freitas validates our implementation of multi-synchronisation by modelling multi-synchronisation and the protocol used in the implementation in *Circus* and proving that these models are related by refinement. A complete formalisation of the strategy, however, is a very complex task, as it involves the semantics of Java and *Circus*. We currently rely on the validation of the implementation of our industrial case study [75], on the implementation of the translator [44], on the fairly direct correspondence of JCSP and *Circus*, and on the trust that JCSP is correctly implemented.

Chapter 7

Conclusion

In this chapter we present an overview of the contributions of our work. Furthermore, related work is also brought into the context of *Circus*. A comparison of *Circus* and these works is provided. Finally, topics for future work are presented.

7.1 Contributions

Recently, researchers have increasingly concentrated their interest in the combination of different existing programming paradigms, which consider different aspects and stages of the software development. Hoare & He did one of the most significant works towards unification [54]. In the *Unifying Theories of Programming*, they use Tarski's relational calculus to give a denotational semantics to constructs from different programming paradigms. Relations between an initial and a subsequent observation of computer devices are used to give specifications, designs, and programs their meanings. Observational variables and associated healthiness conditions characterise theories for imperative, communicating, or sequential processes and their designs.

Following this trend of research, *Circus* [103, 27] combines a model-based language, Z [107], a process algebra, CSP [52], Dijkstra's language of guarded commands [37], and specification statements [63]. It differs from other combinations in that it has an associated refinement theory and calculus.

The *Circus* semantics presented in [30] did not allow us to prove meta-theorems in the *Circus* theory and, as a direct consequence, our refinement laws. For this reason, in Chapter 3, we provided *Circus* with a new and definitive denotational semantics. The approach taken by Cavalcanti and Woodcock [31] was an inspiration for this semantics: we express the semantics of the vast majority of the *Circus* constructs as reactive designs. This uniformity allows us to reuse the results presented in [31]. Together, the work presented in this thesis and the one presented in [31] provide us with a library of lemmas involving reactive designs and foster reuse of these results.

Another important difference between our semantics and the one presented in [30] is the change to the semantics of some of the *Circus* operators. For instance, as a consequence of state changes not resolving choice, in our semantics *Stop* leaves the state loose. Another major difference is the state partitions in the parallel composition and interleaving, which remove the problems intrinsic to shared variables and were suggested in [27]. These partitions also had a direct consequence in the semantics of the parallel composition and interleaving of processes.

Besides the healthiness conditions satisfied by reactive processes (**R1**-**R3**) and by CSP processes (**CSP1**-**CSP3**), *Circus* processes were also proved to satisfy three further healthiness conditions: the first two of them, **C1** and **C2**, have a direct correspondence with two of the extra CSP healthiness conditions, **CSP4** and **CSP5**. However, **C3** is novel; it guarantees that our reactive designs, as one might expect, do not contain any dashed variables in the precondition.

In our way towards a theorem prover for *Circus*, we first gave a set-based model to relations. In order to make the results reusable in the case of a language extension, we have chosen to represent the syntax as functions instead of data types. Based on the theory of relations, we developed four theories: designs, reactive processes, CSP processes, and *Circus* processes. Furthermore, theories that include the declaration and theorems involving the UTP observational variables $okay$, tr, $wait$, and ref were also created and included accordingly in the theories hierarchy, which follows directly from the UTP. With the theory hierarchy we have made the results as general as possible; they can be reused by any other work that has the UTP as its theoretical basis, like, for instance, TCOZ (Timed Communicating Object-Z) [60].

By mechanising the UTP in a theorem prover, we were able to take into account aspects of the UTP that are many times left implicit. The obligation to deal with alphabets makes our work reveal more details on how the alphabets are handled within the UTP. An example is the different use of the existential quantification in different definitions: for variables blocks, it removes the quantified variable from the alphabet, while in the CSP $SKIP$, it does not. Differences were also found in alphabet extension: our model required an alphabet extension which left the value of the new variables unconstrained; in the UTP's alphabet extension, its value cannot change. Finally, some of the healthiness conditions (e.g, **R**) were found to be partial functions, and not total functions as we might expect.

As far as we know, the mechanisation of the *Circus* semantics in ProofPower-Z makes *Circus* the first specification language of concurrent reactive systems that has a mechanised semantics. Based on this result, we intend to mechanise the proof of the refinement laws, providing academia and industry with a mechanised refinement calculus that can be used in the formal development of state-rich reactive programs.

In the UTP, which is the theoretical basis for *Circus*, the notion of refinement is very important. *Circus* adopts for its most basic notion of refinement, action refinement, the UTP definition of refinement: an implementation P satisfies a specification S if, and only if, $[P \Rightarrow S]$. *Circus* processes encapsulate state; hence process refinement is defined in terms of action refinement of the main actions, with the state components taken as local variables. The standard simulation techniques used in Z are adapted to handle processes and data refinement. However, they are slightly different, since actions are total and state initialisation must be explicitly included in the main action: no applicability requirement concerning preconditions exists, and no condition is imposed on the initialisation.

Laws of simulation, and action and process refinements are the basis of the refinement strategy presented in Chapter 4. In this thesis, further laws of simulation and refinement, than those presented in [27], which were needed during the development of the case study presented in Chapter 5, are presented. Furthermore, we present the corrections for some of the laws proposed in [27]; one of the laws proposed in that work was also found to be invalid.

In this thesis, we also prove the correctness of the vast majority of the refinement laws used in our case study. In Chapter 4, we illustrate these proofs and point out the interesting aspects. Some of these proofs were done in terms of other refinement laws. Although not strictly needed, these derived refinement laws provide shortcuts for the users of our method, shortening the development of programs.

Our case study on *Circus*, a safety-critical fire control system, is described in Chapter 5. So far, this is the largest case study on the refinement of *Circus* programs. We have used this case study to verify the usefulness and soundness of the refinement laws discussed in Chapter 4. Using our set of laws, we were able to refine the abstract and centralised specification of the system into a concrete and distributed specification. Furthermore, the case study was also used to analyse the expressiveness of the language. We believe that the use of the refinement strategy in the development of this industrial case study, and in some others [27, 101], provides empirical evidence that the strategy is indeed scalable.

Chapter 6 presented a translation strategy, which makes possible the Java implementation of *Circus* programs. This strategy extends the one published in [73] by including synchronisation and generic channels, indexing operators, generic processes, and multi-synchronisation. It also differs slightly from the one presented in [76]; we present the corrections to the problems pointed out by the work done by Freitas [44]. The main corrections were the introduction of the managers in order to stop the execution of the controllers at the end of the process execution, and the creation of copies of the state components and local variables in both sides of the parallel composition.

In spite of the restrictions on the *Circus* programs discussed in Chapter 6, this translation strategy is already an important tool in the implementation of *Circus* programs, and has been used to implement our case study. Further, it has been used as guideline for a mechanisation of a translation tool presented in [44]. In that work, Freitas also extends our translation strategy by presenting the translation rules for free types, abbreviations, nested external choices and guards, multi-synchronised generic channels, and by allowing actions in a parallel composition to invoke other actions. This tool would have avoided some of the human errors that occurred during the translation of our case study. Hand generated code would be quite simpler; however, some experiments were done and presented almost no performance improvement. More experiments are needed; they become feasible with the availability of the tool.

In summary, this thesis extends and formalises the *Circus* refinement calculus that allows us to develop safety-critical systems following the calculational style of Back [8], Morris [65], and Morgan [63]. Furthermore, it provides a translation strategy from *Circus* to a practical programming language. Together, the refinement calculus and the translation strategy, provide a framework to derive state-rich reactive systems from an abstract specification.

7.2 Related Work

Some other works have already presented the integration of Z or one of its extensions with a process algebra. The main objective of *Circus* is not to be another language like those, but to provide support for the formal development of concurrent programs in a calculational style.

Fischer [40] presents a survey of several integrations of Z with process algebras. Combinations of Z with CCS [46, 93], Z with CSP [82], and Object-Z with CSP [39] are considered in this survey, which also discusses issues involved in the integration of Z with a process algebra. All the approaches above are analysed with respect to these issues.

In [47], Galloway and Stoddart investigate a dialect of value-passing CCS which employs Z as its value calculus. They present a syntactic definition, an example, and finally an operational semantics for the combination. However, using CCS-Z, they are not able to employ the component language CCS in a natural and intuitive way: the syntax of CCS agents are quite complex. For instance, using CCS-Z, we are not able to invoke Z operations simply by name, in a CCS agent, as we do in *Circus*. Besides, no refinement theory was investigated in this combination.

In [82], Woodcock *et al.* give an informal translation from Z to CSP by separating input and output communications. The application of a Z operation is modelled by two CSP events. Fischer generalises this result in his work with CSP-OZ [39], a combination of Object-Z and CSP. CSP-OZ extends Object-Z with the notion of communication channels and CSP syntax. A CSP-OZ specification is composed mostly of class definitions that model processes. They contain an interface (channels definitions), Z schemas that describe the internal state and its initialisation, and CSP processes that model the behaviour of the class. For each channel, an enable schema specifies when communication is possible, and an effect schema specifies the state changes caused by the communication.

Fischer provides a *failures-divergences* semantics for classes with empty CSP parts [39]. Then, the combination of the CSP and Z parts can be simply done by a parallel composition of two classes; one of them contains the interface and the CSP part, and the other one contains the interface and the Z part. This approach reuses existing theories and inherits a lot of theorems. For instance, the monotonicity of the parallel operator allows separate (*failures-divergences*) refinement of the CSP and Z parts. In a different way, Smith [87] gives *failures-divergences* semantics to Object-Z classes, but uses these classes as processes in the CSP part. Furthermore, state-based refinement relations for use on the Object-Z components within an integrated specification have been presented in [89]; these relations, however, do not allow the structure of the specification to be changed in a refinement. In [35], the authors provide such relations, which allow developers to refine the very structure of specifications. Refinement in a calculational style, however, is not considered in any of these works.

TCOZ [60, 61] takes one step further in the combination of CSP and Object-Z: it includes timing primitives by using the timed derivative of CSP [34]. Furthermore, differently from [39] and [87], where operation names take on the role of CSP channels with input and output parameters being passed down the operation channel as values, TCOZ models operations as terminating processes: it accepts a number of inputs, performs a calculation, performs a number of outputs, and terminates. This approach, which is similar to the *Circus* approach, makes it feasible to specify the temporal properties of this calculation when describing the operation, in contrast with [39] and [87], where the atomicity of operations collapses the temporal aspects of operations. The semantics of TCOZ presented in [79], where the authors also present some algebraic laws, blends both timed CSP and Object-Z in a single semantic model based on the UTP framework, as we do in *Circus*. Nevertheless, by using Object-Z, instead of Z, TCOZ compromises a compositional approach to refinement. This is due to Object-Z's reference semantics [88].

Another combination of CSP and Object-Z with real-time constraints is CSP-OZ-DC [55]. Instead of using timed CSP to describe time constraints, as in TCOZ, Hoenicke and Olderog combine Fischer's CSP-OZ with the duration calculus [109], a logic and calculus for specifying real-time systems. The semantics of each CSP-OZ-DC class is given as a timed process; however, only classes with a divergence-free CSP-OZ part are considered. In CSP-OZ-DC, process refinement, data refinement, and time constraint refinement of a system imply the refinement of the whole. A limited reuse of tools like FDR and UPPAAL [13], a model-checker for real-time systems is also possible.

The use of a CSP controller to drive a B machine is the subject of research in CSP $\|$ B [95, 96, 85], where the semantic combination of CSP and B preserves the original semantics of both languages; each description can be analysed separately using the tool support currently available. The style of CSP $\|$ B is akin to Fischer's [39], where each event maps to one operation. In [95], the authors present a method to prove the consistency (in their context, divergence freedom) between a CSP controller and a B machine. They extend these results in [96], where they present a method to ensure deadlock-freedom of these combinations. This result is proved to be compositional in [85], where the authors add assertions and guards in order to prove deadlock and divergence freedom of many controllers and machines, by separately proving that each of the controller-machine pair is divergence free, and that the composition of the controllers (without the machines) is deadlock free. Refinement of controllers and machines can be proved correct using FDR and the B-Toolkit, respectively. However, restrictions on the architecture of CSP controllers and B machines must be enforced in order to make the use of these tools feasible and valid.

Butler and Waldén present an embedding of action systems in the B-method in [20, 18], where they also compare the refinement notions of action systems and B, and suggest extensions to the refinement machines and the proof obligations generated from them for refinement with internal actions (actions that do not affect the global variables of the B machine). Parallel composition of two action systems is introduced as the combination of the disjoint sets of variables and actions. However, without the suggested extensions, the need for extra operations and machines may lead to very complex extra proof conditions. Butler has used these results as an inspiration for csp2b [19], a tool that translates CSP processes into B machines; this translation is justified by an operational semantics. Only a subset of CSP that includes prefix, external choice, and parallel composition and interleaving at the outermost level, is considered.

Abrial presents B# [4], which brings together the notions of refinement, proof, and tools from B, the notion of events from action systems [10], and the notion of generic extension from Z. From B, it also inherits a subset of the mathematical toolkit. The very basic component in a B# system specification is that of a model, which describes a state transition system. States are composed of variables and an invariant, and transitions are events in the form of guarded before-after predicates; there is no conditional action, no explicit choice action, no sequential action, no loop action, and so on. Furthermore, neither a refinement calculus, nor a tool support for it, are available for B#.

In [92], Stoddart uses Z as the basis of an event calculus for communicating state machines, in which communication is modelled by means of shared events that cause a simultaneous change of state in two or more processes. As in *Circus*, machines have their own private data state, which can only be seen by other machines via communication. Although data refinement can be done using standard Z techniques, the refinement of the concurrent aspects does not seem trivial.

Using RSL [50], the RAISE [97] Specification Language, one is also able to describe both data and behavioural aspects of systems. In this language, parallel composition describes communications that may happen, while interlocking describes communications that must happen. External and internal choice, as well as hiding and renaming are also available. Refinement is done using an invent-and-verify method as opposed to a refinement calculus.

The work with action systems is closely related to *Circus*. In an action system, systems are described as a state and a set of guarded commands. Its behaviour is given by a simple interpreter for the program that repeatedly selects an enabled action and executes it. Parallel composition is modelled as the sequential interleaving of atomic steps. Concurrency with shared variables is modelled by partitioning the variables amongst different processes; a model for distributed systems is obtained by partitioning the variables amongst the processes.

The combination of the refinement calculus and action system in the derivation of parallel and distributed algorithms is described in [11]: from a purely sequential algorithm, a stepwise refinement is accomplished until an efficient parallel program is derived. Most steps involve sequential refinements; parallel composition is introduced only through the decomposition of atomic actions.

The very basic nature of the action systems formalism in comparison with process algebra is the main difference between action systems and *Circus*. Action systems have a flat structure, where auxiliary variables simulating program counters are needed to guarantee the proper sequencing of actions. This is due to the simple control flow of action systems: select an enable guard, execute it, repeat. In *Circus*, a richer control flow is provided using CSP operators.

The study of refinement for combinations of Object-Z and CSP has already been undertaken in [90]. Nevertheless, a calculational approach like that of *Circus* has not been adopted. Stepwise refinement for action system has been presented in [11]. However, most of the refinement steps are sequential refinements; in order to introduce parallel composition, one must decompose atomic actions.

Circus adopts the semantic approach to combination: as previously discussed, the UTP is used as the model for *Circus*. The semantic approach adopted by *Circus*, provides a deep integration of the notations. However, with this approach, the semantics of both the Z and CSP operators must be redefined. Fortunately, this is not a major problem for us because we are using an existing semantic model: UTP. Compared to the other combinations presented above, the novelty in *Circus* is the support for refinement of state-rich reactive systems in a calculational style as that presented in [63]. Furthermore, some extensions have already been proposed for *Circus*: a time model for *Circus* has been presented by Sherif and He in [86]; mobile *Circus* processes have been considered in [94]; object-oriented *Circus* is discussed in [24]; and a synchronous *Circus* is currently under investigation.

In [67], the alphabetised relational calculus is formalised in Z/EVES. We extend [67] by including many other operations, such as sequencing, skip, assignment, and nondeterminism. Furthermore, refinement, the complete lattice of relations, and recursion, are part of our work [77].

Besides the work presented in [27], and further case studies, the design rules presented in [69] were another source of inspiration for *Circus* refinement laws. In this work, stepwise development of correct programs is supported by a design calculus for occam-like [58] communicating programs. Specifications are given in terms of assertions. Both program and specifications semantics are uniformly presented in a predicative style similar to that adopted in the UTP.

Lai and Sanders have considered occam in [59], where they present a refinement calculus for communicating processes with states. In their work, they extend an occam-like language with a specification statement in the style of [63]. However, no data refinement method is proposed, and the operators used are limited. Another difference from our work is that they represent deadlock as a special state $\top$, whereas in our model, deadlock is represented as a state in which $wait'$ is $true$.

A refinement calculus for Abstract State Machines is presented in [78] using a style similar to the classical approaches [9, 63] and gives support for the development of ASM specifications. Although possible, the extension of this work in order to deal with communications is left as future work.

In [41], Fischer formalises a translation from CSP-OZ to annotations of Java programs. In the translation, enable and effect schemas become preconditions and postconditions; the CSP part becomes trace assertions, which specify the allowed sequences of method calls; finally, state invariants become class invariants. The result is not an implementation of a CSP-OZ class, but annotations that support the verification of a given implementation. The treatment of class composition is left as future work. Differently, our work supports the translation from *Circus* specifications, possibly describing the interaction between many processes, to correct Java code.

The translation from a subset of CSP-OZ to Java is also considered in [25], where a language called COZJava, which includes CSP-OZ and Java, is used. A CSP-OZ specification is first translated to a description of the structure of the final Java program, which still contains the original CSP processes and Z schemas; these are translated afterwards. The library that they use to implement processes is called CTJ [51], which is in many ways similar to JCSP. The architecture of the resulting Java program is determined by the architecture of CSP-OZ specifications, which keep communications and state update separate. As a consequence, the code is usually inefficient and complicated; this difficulty has motivated the design of *Circus*.

In *Circus*, communications are not attached to state changes, but are freely mixed as exemplified by the action *RegCycle* of process *Register*. As a consequence, the reuse of Z and CSP tools is not straightforward. On the other hand, *Circus* specifications can be refined to code that follow the usual style of programming in languages like occam, or JCSP, and are more efficient.

7.3 Future Work

The work presented in this thesis has provided a means for the formal development of *Circus* specifications into implementable code. However, research on the *Circus* refinement calculus is far from over; a long and rather interesting agenda of research can be envisaged.

The refinement strategy and its laws presented in Chapter 4 are not in the context of the extensions of *Circus*. Refinement calculi for *Circus* time [86], travelling *Circus* [94], Oh*Circus* [24], and synchronous *Circus* still need to be devised. Furthermore, confidentiality aspects of *Circus* is another interesting topic of research.

In Chapter 4, retrieve relations are required to relate every concrete state to some abstract state. Furthermore, the distribution of simulation through a generic external choice required the retrieve relation to be a function from the concrete to the abstract state. In the future, we intend to investigate the completeness of data refinement in *Circus*.

In the calculational approach adopted in this thesis, the repeated application of refinement laws to an abstract specification produces a concrete specification that implements it correctly. However, this may be a hard task, since developments may prove to be long and repetitive. Some development strategies may be captured as sequences of rule applications, and used in different developments, or even several times within a single development. Identifying these strategies, documenting them as refinement tactics in the style of [74], and using them as single transformation rules brings a profit in time and effort. Also, a notation for describing derivations can be used for modifying and analysing formal derivations.

The case studies that have been carried out on the *Circus* refinement calculus give us confidence that the set of laws that are presented in this thesis is appropriate for useful applications. We are aware, however, that the results presented in this thesis are not complete. Nevertheless, in this work, we are concerned with the practicability of refinement, rather than its completeness. In the future, we plan to provide an algebraic semantics for *Circus*, define a normal form, and establish that we have a set of laws that is enough to reduce any terminating *Circus* program to its normal form; this will provide us with a notion of completeness. Furthermore, an investigation on the advantages and consequences on the validity of the laws of refinement for *Circus* specifications that mention the UTP variables is an interesting piece of research that is left as future work.

In Chapter 3, our discussion on alternative models for relations concludes that our restriction on the bindings results in simpler definitions, and hence proofs. This conclusion, however, is based on our experience with ProofPower-Z; some of the alternative models could make proofs easier in another theorem prover. An investigation of alternative theorem provers is a topic for future research.

A natural next step for the mechanisation of the *Circus* semantics is the mechanical proof of the refinement laws proposed in this thesis. This mechanisation is on our agenda of research and will provide *Circus* with a mechanised refinement calculus (and a theorem prover) that can be used in the formal development of state-rich reactive programs.

Some of the hand proofs of the *Circus* refinement laws expand the definition of sequential composition. The mechanical proof of a number of laws for sequential composition will make it possible to avoid these expansions in the mechanisation of the proofs of the refinement laws; it is an important piece of future work.

In the translation strategy presented in Chapter 6 some points still need to be addressed. The first one concerns the translation of iterated parallel composition and interleaving of actions. In the work presented here, we demand the previous extension of such operators before the translation strategy is applied. In order to remove this requirement, we intend to find a way to generalise the solution provided for the simple parallel composition of actions. We also left the investigation into the translation of nested parametrised and indexing processes as future work.

Some of the requirements made concerning the *Circus* programs could be satisfied simply by applying some refinement strategies to the *Circus* programs. Some of these are: a refinement strategy to deal with the restriction on the synchronisation set of channels for parallel composition and interleaving described in Chapter 6 and a refinement strategy for removing output guards.

In Chapter 6, axiomatic definitions of a particular format, free types and abbreviations are considered. Other formats of Z paragraphs still need to be taken into consideration.

In [44], Freitas presents the implementation of a tool that supports our translation strategy; furthermore, she gives the first step towards the validation of the translation strategy by validating our implementation of multi-synchronisation. The complete formalisation of the strategy is a complex task that involves the semantics of Java and *Circus* and is left as future work.

In [45], the author presents the formalisation and implementation of a model-checker for *Circus*; its theoretical basis is the operational semantics for *Circus* presented in [106]. A very interesting piece of future work is to prove the correspondence between our denotational semantics and this operational semantics using the method presented in [54]. Furthermore, the integration of the *Circus* model-checker and our theorem prover presented in Chapter 3, is an adventurous piece of future work. We believe, that together, these tools provide a very powerful framework for formal development of concurrent reactive systems, and for the use of *Circus* refinement calculus in both industry and academia.

Appendix A

Syntax of *Circus*.

$$
\begin{array}{lcl}
\mathsf{Program} & ::= & \mathsf{CircusPar}^{*} \\[4pt]
\mathsf{CircusPar} & ::= & \mathsf{Par} \mid \mathbf{channel}\ \mathsf{CDecl} \mid \mathbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \mid \mathsf{ProcDecl} \\[4pt]
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^{+} \mid \mathsf{N}^{+} : \mathsf{Exp} \mid [\mathsf{N}^{+}]\mathsf{N}^{+} : \mathsf{Exp} \mid \mathsf{SchemaExp} \\
\mathsf{CSExp} & ::= & \{\!|\ |\!\} \mid \{\!|\ \mathsf{N}^{+}\ |\!\} \mid \mathsf{N} \mid \mathsf{CSExp} \cup \mathsf{CSExp} \mid \mathsf{CSExp} \cap \mathsf{CSExp} \\
 & \mid & \mathsf{CSExp} \setminus \mathsf{CSExp} \\[4pt]
\mathsf{ProcDecl} & ::= & \mathbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{ProcDef} \mid \mathbf{process}\ \mathsf{N}[\mathsf{N}^{+}] \mathrel{\widehat=} \mathsf{ProcDef} \\
\mathsf{ProcDef} & ::= & \mathsf{Decl} \bullet \mathsf{ProcDef} \mid \mathsf{Decl} \odot \mathsf{ProcDef} \mid \mathsf{Proc} \\
\mathsf{Proc} & ::= & \mathbf{begin}\ \mathsf{PPar}^{*}\ \mathbf{state}\ \mathsf{SchemaExp}\ \mathsf{PPar}^{*} \bullet \mathsf{Action}\ \mathbf{end} \\
 & \mid & \mathsf{Proc};\ \mathsf{Proc} \mid \mathsf{Proc} \Box \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \\
 & \mid & \mathsf{Proc} \mathbin{|[} \mathsf{CSExp} \mathbin{]|} \mathsf{Proc} \mid \mathsf{Proc} \mathbin{|\!|\!|} \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExp} \\
 & \mid & (\mathsf{Decl} \bullet \mathsf{ProcDef})(\mathsf{Exp}^{+}) \mid \mathsf{N}(\mathsf{Exp}^{+}) \mid \mathsf{N} \\
 & \mid & (\mathsf{Decl} \odot \mathsf{ProcDef})\lfloor \mathsf{Exp}^{+} \rfloor \mid \mathsf{N}\lfloor \mathsf{Exp}^{+} \rfloor \mid \mathsf{Proc}[\mathsf{N}^{+} := \mathsf{N}^{+}] \mid \mathsf{N}[\mathsf{Exp}^{+}] \\
 & \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \Box\, \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \sqcap\, \mathsf{Decl} \bullet \mathsf{Proc} \\
 & \mid & \mathbin{|[} \mathsf{CSExp} \mathbin{]|}\ \mathsf{Decl} \bullet \mathsf{Proc} \mid\ \mathbin{|\!|\!|}\ \mathsf{Decl} \bullet \mathsf{Proc} \\[4pt]
\mathsf{NSExp} & ::= & \{\,\} \mid \{\mathsf{N}^{+}\} \mid \mathsf{N} \mid \mathsf{NSExp} \cup \mathsf{NSExp} \mid \mathsf{NSExp} \cap \mathsf{NSExp} \\
 & \mid & \mathsf{NSExp} \setminus \mathsf{NSExp} \\[4pt]
\mathsf{PPar} & ::= & \mathsf{Par} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{ParAction} \mid \mathbf{nameset}\ \mathsf{N} == \mathsf{NSExp} \\[4pt]
\mathsf{ParAction} & ::= & \mathsf{Action} \mid \mathsf{Decl} \bullet \mathsf{ParAction} \\[4pt]
\mathsf{Action} & ::= & \mathsf{SchemaExp} \mid \mathsf{Command} \mid \mathsf{N} \mid \mathsf{CSPAction} \mid \mathsf{Action}\,[\mathsf{N}^{+} := \mathsf{N}^{+}] \\
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred}\ \&\ \mathsf{Action} \\
 & \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & \mid & \mathsf{Action} \mathbin{|[} \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \mathbin{]|} \mathsf{Action} \\
 & \mid & \mathsf{Action} \mathbin{|\!|[} \mathsf{NSExp} \mid \mathsf{NSExp} \mathbin{]|\!|} \mathsf{Action} \\
 & \mid & \mathsf{Action} \setminus \mathsf{CSExp} \mid \mathsf{ParAction}(\mathsf{Exp}^{+}) \mid \mu\,\mathsf{N} \bullet \mathsf{Action} \\
 & \mid & \fatsemi\ \mathsf{Decl} \bullet \mathsf{Action} \mid\ \Box\, \mathsf{Decl} \bullet \mathsf{Action} \mid\ \sqcap\, \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & \mathbin{|[} \mathsf{CSExp} \mathbin{]|}\ \mathsf{Decl} \bullet \mathbin{|[} \mathsf{NSExp} \mathbin{]|}\ \mathsf{Action} \mid\ \mathbin{|\!|\!|}\ \mathsf{Decl} \bullet \mathbin{|\!|[} \mathsf{NSExp} \mathbin{]|\!|}\ \mathsf{Action} \\[4pt]
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParameter}^{*} \mid \mathsf{N}\ [\mathsf{Exp}^{+}]\ \mathsf{CParameter}^{*} \\
\mathsf{CParameter} & ::= & ?\mathsf{N} \mid ?\mathsf{N} : \mathsf{Pred} \mid !\mathsf{Exp} \mid .\mathsf{Exp}
\end{array}
$$

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

Appendix B

Semantics of *Circus*

B.1 *Circus* Actions

B.1.1 CSP Actions

Definition B.1

$$\begin{aligned} I\!I_{rea} \mathrel{\widehat=}\ & (\neg\, okay \wedge tr \leq tr') \\ & \vee (okay' \wedge tr' = tr \wedge wait' = wait \wedge ref' = ref \wedge v' = v) \end{aligned}$$

Definition B.2 $Stop \mathrel{\widehat=} \ \boldsymbol{R}(true \vdash tr' = tr \wedge wait')$

Definition B.3 $Skip \mathrel{\widehat=} \ \boldsymbol{R}(true \vdash tr' = tr \wedge \neg\, wait' \wedge v' = v)$

Definition B.4 $Chaos \mathrel{\widehat=} \ \boldsymbol{R}(false \vdash true)$

Definition B.5 $A_1;\ A_2 \mathrel{\widehat=} \ A_1 ;_R A_2$

Definition B.6 $g \mathbin{\&} A \mathrel{\widehat=} \ \boldsymbol{R}((g \Rightarrow \neg\, A^f_f) \vdash ((g \wedge A^t_f) \vee (\neg\, g \wedge tr' = tr \wedge wait')))$

Definition B.7

$$A_1 \Box A_2 \mathrel{\widehat=} \ \boldsymbol{R}((\neg\, A_1{}^f_f \wedge \neg\, A_2{}^f_f) \vdash ((A_1{}^t_f \wedge A_2{}^t_f) \lhd tr' = tr \wedge wait' \rhd (A_1{}^t_f \vee A_2{}^t_f)))$$

Definition B.8 $A_1 \sqcap A_2 \mathrel{\widehat=} \ A_1 \vee A_2$

Definition B.9 $do_{\mathcal{C}}\,(c, v) \mathrel{\widehat=} \ tr' = tr \wedge (c, v) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (c, v) \rangle$

Definition B.10 $c \rightarrow Skip \mathrel{\widehat=} \ \boldsymbol{R}(true \vdash do_{\mathcal{C}}\,(c, Sync) \wedge v' = v)$

Definition B.11 $c.e \rightarrow Skip \mathrel{\widehat=} \ \boldsymbol{R}(true \vdash do_{\mathcal{C}}\,(c, e) \wedge v' = v)$

Definition B.12 $c!e \to Skip \mathrel{\widehat=} c.e \to Skip$

Definition B.13 *For any non-input communication* c, $c \to A \mathrel{\widehat=} (c \to Skip); A$

Definition B.14

$$
\begin{array}{rl}
do_{\mathcal{I}}(c, x, P) \mathrel{\widehat=} & tr' = tr \land \{v : \delta(c) \mid P \bullet (c, v)\} \cap ref' = \varnothing \\
& \lhd wait' \rhd \\
& tr' - tr \in \{v : \delta(c) \mid P \bullet \langle (c, v) \rangle\} \land x' = snd(last(tr'))
\end{array}
$$

Definition B.15 $c?x : P \to A(x) \mathrel{\widehat=} \mathbf{var}\ x \bullet \boldsymbol{R}(true \vdash do_{\mathcal{I}}(c, x, P) \land v' = v); A(x)$

Definition B.16 $c?x \to A \mathrel{\widehat=} c?x : true \to A$

Definition B.17 *Multiple data transfer prefix (channels of finite type)*

$$
\begin{array}{l}
c\ cio \to A\ Vars(cio) \mathrel{\widehat=} \\
\qquad \begin{cases} c\ cio \to A\ Vars(cio) & provided\ ?\ is\ not\ in\ cio \\ \Box\ Decls(cio) \bullet c\ Flatten(cio, \epsilon) \to A(Vars(cio)) & otherwise \end{cases}
\end{array}
$$

$$Decls(cio) \mathrel{\widehat=} Sep(Inputs(cio, \langle\rangle),\ ;\ )$$

$$
\begin{array}{l}
Vars(\langle\rangle) \mathrel{\widehat=} \epsilon \\
Vars(cio) \mathrel{\widehat=} (Sep(Names(Inputs(cio, \langle\rangle),\ ,\ )))
\end{array}
$$

$$
\begin{array}{l}
Sep(\langle\rangle, symbol) \mathrel{\widehat=} \langle\rangle \\
Sep(d_1 : \langle d_2 \rangle, symbol) \mathrel{\widehat=} d_1\ symbol\ d_2 \\
Sep(d_1 : ds, symbol) \mathrel{\widehat=} d_1\ symbol\ (SC(ds))
\end{array}
$$

$$
\begin{array}{l}
Inputs(\epsilon, ds) \mathrel{\widehat=} ds \\
Inputs(.e\ cio, ds) \mathrel{\widehat=} Inputs(cio, ds) \\
Inputs(!e\ cio, ds) \mathrel{\widehat=} Inputs(cio, ds) \\
Inputs(?x\ cio, ds) \mathrel{\widehat=} Inputs(cio, ds \frown \langle x : \{x \mid true\} \rangle) \\
Inputs(?x : P\ cio, ds) \mathrel{\widehat=} Inputs(cio, ds \frown \langle x : \{x \mid P\} \rangle)
\end{array}
$$

$$
\begin{array}{l}
Names(\langle\rangle, ns) \mathrel{\widehat=} ns \\
Names(x : ds, ns) \mathrel{\widehat=} (ds, ns \frown \langle x \rangle) \\
Names((x : P) : ds, ns) \mathrel{\widehat=} (ds, ns \frown \langle x \rangle)
\end{array}
$$

$$
\begin{array}{l}
Flatten(\epsilon, rs) \mathrel{\widehat=} rs \\
Flatten(.e\ cio, rs) \mathrel{\widehat=} Flatten(cio, e.Flatten(rs)) \\
Flatten(!e\ cio, rs) \mathrel{\widehat=} Flatten(cio, e.Flatten(rs)) \\
Flatten(?x\ cio, rs) \mathrel{\widehat=} Flatten(cio, x.Flatten(rs)) \\
Flatten(?x : P\ cio, rs) \mathrel{\widehat=} Flatten(cio, x.Flatten(rs))
\end{array}
$$

Definition B.18

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 \mathrel{\widehat=}$$
$$\boldsymbol{R}\left(\begin{array}{l} \neg\ \exists 1.tr', 2.tr' \bullet ({A_1}^f_f;\ 1.tr' = tr) \land ({A_2}_f;\ 2.tr' = tr) \\ \qquad\qquad\qquad\quad \land\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\ \land \neg\ \exists 1.tr', 2.tr' \bullet ({A_1}_f;\ 1.tr' = tr) \land ({A_2}^f_f;\ 2.tr' = tr) \\ \qquad\qquad\qquad\quad \land\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\ \vdash \\ (({A_1}^t_f;\ U1(out\alpha\, A_1)) \land ({A_2}^t_f;\ U2(out\alpha\, A_2)))_{+\{v,tr\}};\ M_{\parallel_{cs}} \end{array}\right)$$

$$U1(v'_1, \ldots, v'_n) \mathrel{\widehat=}\ 1.v'_1 = v_1 \land \ldots \land 1.v'_n = v_n$$
$$\alpha U1(v'_1, \ldots, v'_n) = \{1.v'_1, \ldots, 1.v'_n, v_1, \ldots, v_n\}$$

$$U2(v'_1, \ldots, v'_n) \mathrel{\widehat=}\ 2.v'_1 = v_1 \land \ldots \land 2.v'_n = v_n$$
$$\alpha U2(v'_1, \ldots, v'_n) = \{2.v'_1, \ldots, 2.v'_n, v_1, \ldots, v_n\}$$

$$\begin{array}{ll} MSt \mathrel{\widehat=} & \forall\, v \bullet v \in ns_1 \Rightarrow v' = 1.v \\ & \qquad \land\ v \in ns_2 \Rightarrow v' = 2.v \\ & \qquad \land\ v \notin ns_1 \cup ns_2 \Rightarrow v' = v \end{array}$$

$$\begin{array}{ll} M_{\parallel_{cs}} \mathrel{\widehat=} & tr' - tr \in (1.tr - tr \parallel_{cs} 2.tr - tr) \\ & \land\ 1.tr \upharpoonright cs = 2.tr \upharpoonright cs \\ & \land \left(\begin{array}{l} \left(\begin{array}{l} (1.wait \lor 2.wait) \\ \land\ ref' \subseteq ((1.ref \cup 2.ref) \cap cs) \cup ((1.ref \cap 2.ref) \setminus cs) \end{array}\right) \\ \lhd wait' \rhd \\ (\neg\ 1.wait \land \neg\ 2.wait \land MSt) \end{array}\right) \end{array}$$

$$\langle\rangle \parallel_{cs} \langle\rangle \mathrel{\widehat=}\ \{\langle\rangle\}$$
$$e : tr \parallel_{cs} \langle\rangle \mathrel{\widehat=}\ \{\langle\rangle\} \lhd e \in cs \rhd \{x \mid hd(x) = e \land tl(x) \in (tr \parallel_{cs} \langle\rangle)\}$$
$$\langle\rangle \parallel_{cs} e : tr \mathrel{\widehat=}\ \{\langle\rangle\} \lhd e \in cs \rhd \{x \mid hd(x) = e \land tl(x) \in (\langle\rangle \parallel_{cs} tr)\}$$
$$\begin{array}{l} e : tr_1 \parallel_{cs} e : tr_2 \mathrel{\widehat=} \\ \quad \{x \mid hd(x) = e \land tl(x) \in (tr_1 \parallel_{cs} tr_2)\} \\ \quad \lhd e \in cs \rhd \\ \quad \left(\begin{array}{l} \{x \mid hd(x) = e \land tl(x) \in (tr_1 \parallel_{cs} e : tr_2)\} \\ \cup \\ \{x \mid hd(x) = e \land tl(x) \in (e : tr_1 \parallel_{cs} tr_2)\} \end{array}\right) \end{array}$$

$$\begin{array}{l} e_1 : tr_1 \parallel_{cs} e_2 : tr_2 \mathrel{\widehat=} \\ \quad \left(\begin{array}{l} \{\langle\rangle\} \\ \lhd e_2 \in cs \rhd \\ \{x \mid hd(x) = e_2 \land tl(x) \in (e_1 : tr_1 \parallel_{cs} tr_2)\} \end{array}\right) \\ \quad \lhd e_1 \in cs \rhd \\ \quad \left(\begin{array}{l} \{x \mid hd(x) = e_1 \land tl(x) \in (tr_1 \parallel_{cs} e_2 : tr_2)\} \\ \lhd e_2 \in cs \rhd \\ \left(\begin{array}{l} \{x \mid hd(x) = e_1 \land tl(x) \in (tr_1 \parallel_{cs} e_2 : tr_2)\} \\ \cup \\ \{x \mid hd(x) = e_2 \land tl(x) \in (e_1 : tr_1 \parallel_{cs} tr_2)\} \end{array}\right) \end{array}\right) \end{array}$$

Definition B.19

$$
\begin{array}{l}
A_1 \; [\![ns_2 \mid ns_2]\!] \; A_2 \mathrel{\widehat=} \\
\quad \boldsymbol{R}\left(\begin{array}{l}
(\neg\, {A_1}^f_f \wedge \neg\, {A_2}^f_f) \\
\vdash \\
(({A_1}^t_f;\ U1(out\alpha\, A_1)) \wedge ({A_2}^t_f;\ U2(out\alpha\, A_2)))_{+\{v,tr\}};\ M_{|||_{cs}}
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
M_{|||} \mathrel{\widehat=} \; tr' - tr \in (1.tr - tr \; ||| \; 2.tr - tr) \\
\qquad \wedge \left(\begin{array}{l}
((1.wait \vee 2.wait) \wedge ref' \subseteq 1.ref \cap 2.ref) \\
\lhd wait' \rhd \\
(\neg\, 1.wait \wedge \neg\, 2.wait \wedge MSt)
\end{array}\right)
\end{array}
$$

$$
\begin{array}{l}
\langle\rangle \; ||| \; \langle\rangle \mathrel{\widehat=} \; \{\langle\rangle\} \\
tr_1 \; ||| \; \langle\rangle \mathrel{\widehat=} \; \{tr_1\} \\
\langle\rangle \; ||| \; tr_2 \mathrel{\widehat=} \; \{tr_2\} \\
e_1 : tr_1 \; ||| \; e_2 : tr_2 \mathrel{\widehat=} \; \begin{array}[t]{l}
\{x \mid hd(x) = e_1 \wedge tl(x) \in (tr_1 \; ||| \; e_2 : tr_2)\} \\
\cup \\
\{x \mid hd(x) = e_2 \wedge tl(x) \in (e_1 : tr_1 \; ||| \; tr_2)\}
\end{array}
\end{array}
$$

Definition B.20

$$
\begin{array}{l}
A \setminus cs \mathrel{\widehat=} \\
\quad \boldsymbol{R}(\exists\, s \bullet A[s, cs \cup ref'/tr', ref'] \wedge (tr' - tr) = (s - tr) \upharpoonright (EVENT - cs));\ Skip
\end{array}
$$

Definition B.21 $\mu\, X \bullet F(X) \mathrel{\widehat=} \; \bigsqcap \{X \mid F(X) \sqsubseteq_{\mathcal{A}} X\}$

Iterated Operators

Definition B.22 $\fatsemi\; x : \langle v_1, \ldots, v_n\rangle \bullet A(x) \mathrel{\widehat=} \; A(v_1);\ \ldots;\ A(v_n)$

Definition B.23 $\Box\, x : T \bullet A(x) \mathrel{\widehat=} \; A(v_1) \,\Box\, \ldots \,\Box\, A(v_n)$

Definition B.24 $\sqcap\, x : T \bullet A(x) \mathrel{\widehat=} \; A(v_1) \sqcap \ldots \sqcap A(v_n)$

Definition B.25

$$
\begin{array}{l}
[\![\, cs \,]\!] \; x : \{v_1, \ldots, v_n\} \bullet [\![\, ns(x) \,]\!] \; A(x) \mathrel{\widehat=} \; \begin{array}[t]{l}
A(v_1) \\
[\![ns(v_1) \mid cs \mid \bigcup\{x : \{v_2, \ldots, v_n\} \bullet ns(x)\}]\!] \\
\left(\ldots \left(\begin{array}{l}
A(v_{n-1}) \\
[\![ns(v_{n-1}) \mid cs \mid ns(v_n)]\!] \\
A(v_n)
\end{array}\right)\right)
\end{array}
\end{array}
$$

Definition B.26

$$
\begin{array}{l}
||| \; x : \{v_1, \ldots, v_n\} \bullet [\![\, ns(x) \,]\!] \; A(x) \mathrel{\widehat=} \; \begin{array}[t]{l}
A(v_1) \\
[\![ns(v_1) \mid \bigcup\{x : \{v_2, \ldots, v_n\} \bullet ns(x)\}]\!] \\
\left(\ldots \left(\begin{array}{l}
A(v_{n-1}) \\
[\![ns(v_{n-1}) \mid ns(v_n)]\!] \\
A(v_n)
\end{array}\right)\right)
\end{array}
\end{array}
$$

B.1.2 Action Invocations, Parametrised Actions and Renaming

In what follows, we consider the function B, which gives us the body of the action, given its name.

Definition B.27 $N \mathrel{\widehat{=}} \mathrm{B}(N)$

Definition B.28 $N(e) \mathrel{\widehat{=}} \mathrm{B}(N)(e)$

Definition B.29 $(x : T \bullet A)(e) \mathrel{\widehat{=}} A[e/x]$

Definition B.30

$$\begin{array}{l} A[old_1, \ldots, old_n := new_1, \ldots, new_n] \\ \widehat{=} \\ A[old_1, \ldots, old_n / new_1, \ldots, new_n] \end{array}$$

B.1.3 Commands

Definition B.31

$$\begin{array}{l} x_1, \ldots, x_n := e_1, \ldots, e_n \mathrel{\widehat{=}} \\ \quad \boldsymbol{R}(true \vdash tr' = tr \land \neg\, wait' \land x_1' = e_1 \land \ldots \land x_n' = e_n \land u' = u) \end{array}$$

Definition B.32 $w : [\, pre, post \,] \mathrel{\widehat{=}} \boldsymbol{R}(pre \vdash post \land \neg\, wait' \land tr' = tr \land u' = u)$

Definition B.33 $\{g\} \mathrel{\widehat{=}} : [g, true]$

Definition B.34 $[g] \mathrel{\widehat{=}} : [g]$

Definition B.35

$$\mathbf{if} \, [\!] \; i \bullet g_i \rightarrow A_i \; \mathbf{fi} \mathrel{\widehat{=}} \boldsymbol{R}((\bigvee i \bullet g_i) \land (\bigwedge i \bullet g_i \Rightarrow \neg\, A_{i_f}^{\,f}) \vdash \bigvee i \bullet (g_i \land A_{i_f}^{\,t}))$$

Definition B.36 $\mathbf{var}\; x : T \bullet A \mathrel{\widehat{=}} \mathbf{var}\; x : T;\, A;\, \mathbf{end}\; x : T$

Definition B.37

$$(\mathbf{val}\; x : T \bullet A)(e) \mathrel{\widehat{=}} (\mathbf{var}\; x : T \bullet x := e;\, A)$$

provided $x \notin FV(e)$

Definition B.38 $(\mathbf{res}\; x : T \bullet A)(y) \mathrel{\widehat{=}} (\mathbf{var}\; x : T \bullet A;\, y := x)$

Definition B.39

$$(\mathbf{vres}\; x : T \bullet A)(y) \mathrel{\widehat{=}} (\mathbf{var}\; x : T \bullet x := y;\, A;\, y := x)$$

provided $x \neq y$

B.1.4 Schema Expressions

Definition B.40 $[udecl;\ ddecl' \mid pred] \mathrel{\widehat=} ddecl : [\exists\, ddecl' \bullet pred, pred]$

B.2 *Circus* Processes

Definition B.41

$$\textbf{begin state}\ [decl \mid pred]\ PPars\ \bullet A\ \textbf{end} \mathrel{\widehat=} \textbf{var}\ decl \bullet A$$

Definition B.42 *For* $op \in \{\, ;\, , \Box\, , \sqcap \}$*:*

$$\begin{array}{ll} P\ op\ Q \mathrel{\widehat=} & \textbf{begin state}\ State \mathrel{\widehat=} P.State \land Q.State \\ & \quad P.PPar \land_{\Xi} Q.State \\ & \quad Q.PPar \land_{\Xi} P.State \\ & \quad \bullet\ P.Act\ op\ Q.Act \\ & \textbf{end} \end{array}$$

Definition B.43

$$\begin{array}{ll} P \mathbin{[\![} cs \mathbin{]\!]} Q \mathrel{\widehat=} & \textbf{begin state}\ State \mathrel{\widehat=} P.State \land Q.State \\ & \quad P.PPar \land_{\Xi} Q.State \\ & \quad Q.PPar \land_{\Xi} P.State \\ & \quad \bullet\ P.Act \mathbin{[\![} \alpha(P.State) \mid cs \mid \alpha(Q.State) \mathbin{]\!]} Q.Act \\ & \textbf{end} \end{array}$$

Definition B.44

$$\begin{array}{ll} P \interleave Q \mathrel{\widehat=} & \textbf{begin state}\ State \mathrel{\widehat=} P.State \land Q.State \\ & \quad P.PPar \land_{\Xi} Q.State \\ & \quad Q.PPar \land_{\Xi} P.State \\ & \quad \bullet\ P.Act \mathbin{|\!|\![} \alpha(P.State) \mid \alpha(Q.State) \mathbin{]\!|\!|} Q.Act \\ & \textbf{end} \end{array}$$

Definition B.45 $P \setminus cs \mathrel{\widehat=} \textbf{state}\ State \mathrel{\widehat=} P.State\ P.PPar\ \bullet P.Act \setminus cs\ \textbf{end}$

Definition B.46 $x : T \odot P \mathrel{\widehat=} (x : T \bullet P)[c : usedC(P) \bullet c\_x.x]$

Definition B.47 $(x : T \odot P)\lfloor v \rfloor \mathrel{\widehat=} (x : T \odot P)(v)$

Definition B.48 $N\lfloor v_1 \rfloor \mathrel{\widehat=} \mathrm{B}(P)\lfloor v \rfloor$

Definition B.49 $N \mathrel{\widehat=} \mathrm{B}(N)$

Definition B.50 $N(e) \mathrel{\widehat=} \mathrm{B}(N)(e)$

Definition B.51 $(x : T \bullet P)(e) \mathrel{\widehat=} P[e/x]$

Definition B.52 $\mathbin{;}\, x : \langle v_1, \ldots, v_n \rangle \bullet P(x) \mathrel{\widehat=}\ P(v_1);\ \ldots;\ P(v_n)$

Definition B.53 $\Box\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat=}\ P(v_1) \Box \ldots \Box P(v_n)$

Definition B.54 $\sqcap\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat=}\ P(v_1) \sqcap \ldots \sqcap P(v_n)$

Definition B.55 $\llbracket cs \rrbracket\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat=}\ P(v_1) \llbracket cs \rrbracket (\ldots (P(v_{n-1}) \llbracket cs \rrbracket P(v_n)))$

Definition B.56 $|||\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat=}\ P(v_1) ||| (\ldots (P(v_{n-1}) ||| P(v_n)))$

Definition B.57 $P[oldc := newc] \mathrel{\widehat=}\ P[newc/oldc]$

In what follows, we consider the function I, which instantiates the Z paragraphs and channels within a generic process (declared using generic parameters $T_0, \ldots, T_n$) with the types that are given.

Definition B.58 $P[te_0, \ldots, te_n] \mathrel{\widehat=}\ \mathrm{I}(\mathrm{B}(P), \langle te_0, \ldots, te_n \rangle)$

Appendix C

Refinement Laws

Simulation

Law C.1 (*Skip*)

$$Skip \preceq Skip$$

Law C.2 (*Stop*)

$$Stop \preceq Stop$$

Law C.3 (*Chaos*)

$$Chaos \preceq Chaos$$

Law C.4 (Schema expressions)

$$ASExp \preceq CSExp$$

provided

- $\forall P_1.State;\ P_2.State;\ L \bullet R \land \text{pre } ASExp \Rightarrow \text{pre } CSExp$
- $\forall P_1.State;\ P_2.State;\ P_2.State';\ L \bullet R \land \text{pre } ASExp \land CSExp \Rightarrow$ $(\exists P_1.State';\ L' \bullet R' \land ASExp)$ ❑

Law C.5 (Prefix distribution*)

$$c \rightarrow A_1 \preceq c \rightarrow A_2$$

provided $A_1 \preceq A_2$ ❑

Law C.6 (Simple prefix distribution*)

$$c.ae \rightarrow Skip \preceq c.ce \rightarrow Skip$$

provided

- ➩ $\forall P_1.State;\ P_2.State;\ L \bullet R \Rightarrow ae = ce$ ❑

Law C.7 (Output prefix distribution)

$$c!ae \rightarrow A_1 \preceq c!ce \rightarrow A_2$$

provided

- ➩ $\forall P_1.State;\ P_2.State;\ L \bullet R \Rightarrow ae = ce$
- ➩ $A_1 \preceq A_2$ ❑

Law C.8 (Input prefix distribution)

$$c?x \rightarrow A_1 \preceq c?x \rightarrow A_2$$

provided $A_1 \preceq A_2$ ❑

Law C.9 (Input constrained prefix distribution*)

$$c?x : T_1 \rightarrow A_1 \preceq c?x : T_1 \rightarrow A_2$$

provided

- ➩ $A_1 \preceq A_2$
- ➩ $\forall A_1.State;\ A_2.State;\ L \bullet R \Rightarrow (T_1 \Leftrightarrow T_2)$ ❑

Law C.10 (Multiple prefix distribution*)
For every channel c and communication parameters as and cs,

$$c\ as \rightarrow A_1 \preceq c\ cs \rightarrow A_2$$

provided

- ➩ $A_1 \preceq A_2$
- ➩ *For every abstract expression e_{a_i} in as and its corresponding concrete expression e_{c_i} in cs:* $\forall P_1.State;\ P_2.State;\ L \bullet R \Rightarrow (e_{a_i} \Leftrightarrow e_{a_i})$
- ➩ *The names of all input variables are not changed from as to cs.*
- ➩ *Type of c is finite.* ❑

Law C.11 (Guard distribution)

$$ag \mathbin{\&} A_1 \preceq cg \mathbin{\&} A_2$$

provided

- ➩ $\forall P_1.State;\ P_2.State;\ L \bullet R \Rightarrow (ag \Leftrightarrow cg)$
- ➩ $A_1 \preceq A_2$ ❑

Law C.12 (Sequence distribution)

$$A_1;\ A_2 \preceq B_1;\ B_2$$

provided

- ➩ $A_1 \preceq B_1$
- ➩ $A_2 \preceq B_2$ ❑

Law C.13 (External choice distribution*)

$$A_1 \Box A_2 \preceq B_1 \Box B_2$$

provided

- ➩ $A_1 \preceq B_1$
- ➩ $A_2 \preceq B_2$
- ➩ *R is a function from the concrete to the abstract state* ❑

Law C.14 (External choice/Prefix distribution*)

$$\Box\ i \bullet c_i \to A_i \preceq \Box\ i \bullet c_i \to B_i$$

provided $\forall i \bullet A_i \preceq B_i$ ❑

Law C.15 (External choice/Simple prefix distribution*)

$$\Box\ i \bullet c_i.ae_i \to A_i \preceq \Box\ i \bullet c_i.ce_i \to B_i$$

provided

- ➩ $\forall i \bullet A_i \preceq B_i$
- ➩ $\forall i \bullet \forall P_1.State;\ P_2.State;\ L \bullet R \Rightarrow ae_i = ce_i$ ❑

Law C.16 (External choice/Output Prefix distribution*)

$$\Box\, i \bullet c_i!ae_i \rightarrow A_i \preceq \Box\, i \bullet c_i!ce_i \rightarrow B_i$$

provided

- $\forall\, i \bullet A_i \preceq B_i$
- $\forall\, i \bullet \forall\, P_1.State;\ P_2.State;\ L \bullet R \Rightarrow ae_i = ce_i$ ❏

Law C.17 (External choice/Input Prefix distribution*)

$$\Box\, i \bullet c_i?x_i \rightarrow A_i \preceq \Box\, i \bullet c_i?x_i \rightarrow B_i$$

provided $\forall\, i \bullet A_i \preceq B_i$ ❏

Law C.18 (External choice/Constrained Input Prefix distribution*)

$$\Box\, i \bullet c_i?x_i : T_{A_i} \rightarrow A_i \preceq \Box\, i \bullet (c_i?x_i : T_{B_i} \rightarrow B_i$$

provided

- $\forall\, i \bullet A_i \preceq B_i$
- $\forall\, i \bullet \forall\, A.State;\ B.State;\ L \bullet R \Rightarrow (T_{A_i} \Leftrightarrow T_{B_i})$ ❏

Law C.19 (External choice/Multiple Prefix distribution*)
For every channel c_i and communication parameters as_i, and cs_i,

$$\Box\, i \bullet c_i\ as_i \rightarrow A_i \preceq \Box\, i \bullet c_i\ cs_i \rightarrow B_i$$

provided

- *Type of c is finite*
- $\forall\, i \bullet A_i \preceq B_i$
- *For every i, and every abstract expression e_a in as_i and its corresponding concrete expression*
  e_c in cs_i: $\forall\, P_1.State;\ P_2.State;\ L \bullet R \Rightarrow e_a \Leftrightarrow e_c$
- *For every i, the names of all input variables are not changed neither from as_i to cs_i* ❏

Law C.20 (Internal choice distribution*)

$$A_1 \sqcap A_2 \preceq B_1 \sqcap B_2$$

provided

- $A_1 \preceq A_2$
- $B_1 \preceq B_2$ ❏

Law C.21 (Parallelism composition distribution*)

$$A_1 \llbracket\, ns_{1_A} \mid cs \mid ns_{2_A} \,\rrbracket\, A_2 \;\preceq\; B_1 \llbracket\, ns_{1_B} \mid cs \mid ns_{2_B} \,\rrbracket\, B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$
- $\forall\, v_A, v_B \bullet R(v_A, v_B) \Rightarrow (v_A \in ns_{1_A} \Rightarrow v_B \in ns_{1_B})$
- $\forall\, v_A, v_B \bullet R(v_A, v_B) \Rightarrow (v_A \in ns_{2_A} \Rightarrow v_B \in ns_{2_B})$ ❑

Law C.22 (Interleave distribution*)

$$A_1 \;|||[ns_1 \mid ns_2]|||\; A_2 \;\preceq\; B_1 \;|||[ns_1 \mid ns_2]|||\; B_2$$

provided

- $A_1 \preceq A_2$
- $B_1 \preceq B_2$
- $\forall\, v_A, v_B \bullet R(v_A, v_B) \Rightarrow (v_A \in ns_{1_A} \Rightarrow v_B \in ns_{1_B})$
- $\forall\, v_A, v_B \bullet R(v_A, v_B) \Rightarrow (v_A \in ns_{2_A} \Rightarrow v_B \in ns_{2_B})$ ❑

Law C.23 (Recursion distribution*)

$$\mu\, X \bullet F_A(X) \;\preceq\; \mu\, X \bullet F_C(X)$$

provided $F_A \preceq F_C$ ❑

Law C.24 (Specification Statement Distribution*)

$$w_A : [pre_A, post_A] \;\preceq\; w_B : [pre_B, post_B]$$

provided

- $\neg\, pre_A \preceq \neg\, pre_B$
- $post_A \wedge u'_A = u_A \preceq post_B \wedge u'_B = u_B$, *where u are the state variables that are not in the frame w.* ❑

Law C.25 (Variable Block Distribution*)

$$\mathbf{var}\ x \bullet A_1 \;\preceq\; \mathbf{var}\ x \bullet A_2$$

provided

$A_1 \preceq A_2$ ❑

Action Refinement

Assumptions

Law C.26 (Assumption Conjunction*)

$$\{g_1\};\ \{g_2\} = \{g_1 \wedge g_2\}$$

Law C.27 (Assumption introduction*)

$$\{g\} = \{g\};\ \{g_1\}$$

provided $g \Rightarrow g_1$ ❑

In the following two laws we refer to a predicate $assump'$. In general, for any predicate p, the predicate p' is formed by dashing all its free undecorated variables.

Law C.28 (Schema Expression/Assumption—introduction)

$$\begin{array}{l}[\Delta State;\ i? : T_i;\ o! : T_o \mid p \wedge assump'] \\ = \\ [\Delta State;\ i? : T_i;\ o! : T_o \mid p \wedge assump'];\ \{assump\}\end{array}$$

The schema in this law is an arbitrary schema that specifies an action in *Circus*: it acts on a state schema *State* and, optionally, has input variables $i?$ of type T_i, and output variables $o!$ of type T_o.

Law C.29 (Initialisation schema/Assumption—introduction*)

$$\begin{array}{l}[State' \mid p \wedge assump'] \\ = \\ [State' \mid p \wedge assump'];\ \{assump\}\end{array}$$

Law C.30 (Assumption/Guard—introduction)

$$\{\, g \,\};\ A = \{\, g \,\};\ g \ \&\ A$$

Law C.31 (Guard/Assumption—introduction 1*)

$$g \ \&\ A = g \ \&\ \{g\};\ A$$

Law C.32 (Assumption/Guard—elimination 1)

$$\{\, g_1 \,\};\ (g_2 \ \&\ A) = \{\, g_1 \,\};\ A$$

provided $g_1 \Rightarrow g_2$ ❑

Law C.33 (Assumption/Guard—elimination 2)

$$\{ g_1 \};\ (g_2 \mathbin{\&} A) = \{ g_1 \};\ \mathit{Stop}$$

provided $g_1 \Rightarrow \neg\, g_2$ ❑

Law C.34 (Assumption/Guard—replacement)

$$\{ g_1 \};\ (g_2 \mathbin{\&} A) = \{ g_1 \};\ (g_3 \mathbin{\&} A)$$

provided $g_1 \Rightarrow (g_2 \Leftrightarrow g_3)$ ❑

Law C.35 (Assumption elimination)

$$\{p\} \sqsubseteq_{\mathcal{A}} \mathit{Skip}$$

Law C.36 (Assumption substitution 1*)

$$\{g_1\} \sqsubseteq_{\mathcal{A}} \{g_2\}$$

provided $g_1 \Rightarrow g_2$ ❑

Law C.37 (Assumption/External choice—distribution)

$$\{p\};\ (A_1 \Box A_2) = (\{p\};\ A_1) \Box (\{p\};\ A_2)$$

Law C.38 (Assumption/Parallelism composition—distribution)

$$\{p\};\ (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2) = (\{p\};\ A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, (\{p\};\ A_2)$$

Law C.39 (Assumption/Interleaving—distribution)

$$\{p\};\ (A_1 \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} A_2) = (\{p\};\ A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (\{p\};\ A_2)$$

Law C.40 (Assumption/Mutual recursion—distribution*)

$$\{g\};\ \mu\, X_1, \ldots, X_i, \ldots, X_n \bullet \left\langle \begin{array}{l} F_1(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_i(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_n(X_1, \ldots, X_i, \ldots, X_n) \end{array} \right\rangle$$

$$\sqsubseteq_{\mathcal{A}}$$

$$\mu\, X_1, \ldots, X_i, \ldots, X_n \bullet \left\langle \begin{array}{l} F_1(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ \{g\};\ F_i(X_1, \ldots, X_i, \ldots, X_n), \ldots, \\ F_n(X_1, \ldots, X_i, \ldots, X_n) \end{array} \right\rangle$$

provided *for all j, such that $1 \leq j \leq n$,*

$$\{g\};\ F_j(X_1, \ldots, X_i, \ldots, X_n) \sqsubseteq_{\mathcal{A}} F_j(\{g\};\ X_1, \ldots, \{g\};\ X_i, \ldots, \{g\};\ X_n),$$

❑

Law C.41 (Assumption/Prefix—distribution*)

$$\{g\};\ c \rightarrow A \sqsubseteq_{\mathcal{A}} c \rightarrow \{g\};\ A$$

Law C.42 (Assumption/Prefix—distribution 2*)

$$\{g\};\ c \rightarrow A = \{g\};\ c \rightarrow \{g\};\ A$$

Law C.43 (Assumption/Simple Prefix—distribution*)

$$\{g\};\ c.e \rightarrow A \sqsubseteq_{\mathcal{A}}\ ;\ c.e \rightarrow \{g\};\ A$$

Law C.44 (Assumption/Simple Prefix—distribution 2*)

$$\{g\};\ c.e \rightarrow A = \{g\};\ c.e \rightarrow \{g\};\ A$$

Law C.45 (Assumption/Output prefix—distribution*)

$$\{g\};\ c!x \rightarrow A \sqsubseteq_{\mathcal{A}} c!x \rightarrow \{g\};\ A$$

Law C.46 (Assumption/Output prefix—distribution 2*)

$$\{g\};\ c!x \rightarrow A = \{g\};\ c!x \rightarrow \{g\};\ A$$

Law C.47 (Assumption/Input prefix—distribution*)

$$\{g\};\ c?x \rightarrow A \sqsubseteq_{\mathcal{A}} c?x \rightarrow \{g\};\ A$$

provided $x \notin FV(g)$ ❑

Law C.48 (Assumption/Input Prefix—distribution 2*)

$$\{g\};\ c?x \rightarrow A = \{g\};\ c?x \rightarrow \{g\};\ A$$

provided $x \notin FV(g)$ ❑

Law C.49 (Assumption/Constrained Input prefix—distribution*)

$$\{g\};\ c?x : T \rightarrow A \sqsubseteq_{\mathcal{A}} c?x : T \rightarrow \{g\};\ A$$

provided $x \notin FV(g)$ ❑

Law C.50 (Assumption/Constrained Input Prefix—distribution 2*)

$$\{g\};\ c?x : T \rightarrow A = \{g\};\ c?x : T \rightarrow \{g\};\ A$$

provided $x \notin FV(g)$ ❑

Law C.51 (Assumption/Multiple prefix—distribution*)
For every channel c and communication parameters as,

$$\{g\};\ c\ as \rightarrow A \sqsubseteq_{\mathcal{A}} c\ as \rightarrow \{g\};\ A$$

provided

➩ *The names of all input variables are not free in g.* ❑

Law C.52 (Assumption/Multiple Prefix—distribution 2*)
For every channel c and communication parameters as,

$$\{g\};\ c\ as \rightarrow A = c\ as \rightarrow \{g\};\ A$$

provided

➩ *The names of all input variables are not free in g.* ❑

Law C.53 (Assumption/Schema—distribution*)

$$\{g\};\ [decl \mid p] \sqsubseteq_{\mathcal{A}} [decl \mid p];\ \{g\}$$

provided $g \land p \Rightarrow g'$ ❑

Law C.54 (Assumption/Assignment—distribution*)

$$\{g\};\, x := e = \{g\};\, x := e;\, \{g\}$$

provided $x \notin FV(g)$ ❑

Law C.55 (Assumption Unit*)

$$\{true\} = Skip$$

Law C.56 (Assumption Zero*)

$$\{false\} = Chaos$$

Guards

Law C.57 (Guard combination)

$$g_1 \,\&\, (g_2 \,\&\, A) = (g_1 \wedge g_2) \,\&\, A$$

Law C.58 (Guards expansion*)

$$(g_1 \vee g_2) \,\&\, A = g_1 \,\&\, A \,\Box\, g_2 \,\&\, A$$

Law C.59 (Guard/Sequence—associativity)

$$(g \,\&\, A_1);\ A_2 = g \,\&\, (A_1;\ A_2)$$

Law C.60 (Guard/External choice—distribution)

$$g \,\&\, (A_1 \,\Box\, A_2) = (g \,\&\, A_1) \,\Box\, (g \,\&\, A_2)$$

Law C.61 (Guard/Internal choice—distribution)

$$g \,\&\, (A_1 \sqcap A_2) = (g \,\&\, A_1) \sqcap (g \,\&\, A_2)$$

Law C.62 (Guard/Parallelism composition—distribution 1)

$$g \,\&\, (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (g \,\&\, A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g \,\&\, A_2)$$

Law C.63 (Guard/Parallelism composition—distribution 2)

$$(g_1 \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \mathbin{\&} A_2)$$
$$=$$
$$(g_1 \lor g_2) \mathbin{\&} ((g_1 \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \mathbin{\&} A_2))$$

provided

➩ $g_1 \Leftrightarrow g_2$ ❑

Law C.64 (Guards/Parallelism composition—distribution 3*)

$$(g_1 \land g_2) \mathbin{\&} (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$$
$$=$$
$$(g_1 \mathbin{\&} A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (g_2 \mathbin{\&} A_2)$$

provided

➩ $g_1 \Leftrightarrow g_2$ ❑

Law C.65 (Guard/Interleaving—distribution 1)

$$g \mathbin{\&} (A_1 \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} A_2) = (g \mathbin{\&} A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (g \mathbin{\&} A_2)$$

Law C.66 (Guard/Interleaving—distribution 2)

$$(g_1 \mathbin{\&} A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (g_2 \mathbin{\&} A_2)$$
$$=$$
$$(g_1 \lor g_2) \mathbin{\&} ((g_1 \mathbin{\&} A_1) \mathbin{|||}[ns_1 \mid ns_2]\mathbin{|||} (g_2 \mathbin{\&} A_2))$$

Law C.67 (True guard)

$$true \mathbin{\&} A = A$$

Law C.68 (False guard)

$$false \mathbin{\&} A = Stop$$

Law C.69 (Guarded *Stop*)

$$g \mathbin{\&} Stop = Stop$$

Schema Expressions

Law C.70 (Schema disjunction elimination)

$$\text{pre } SExp_1 \mathbin{\&} (SExp_1 \lor SExp_2) \ \sqsubseteq_{\mathcal{A}} \ \text{pre } SExp_1 \mathbin{\&} SExp_1$$

Law C.71 (Schema expression/Sequence—introduction)

$$[\Delta S_1;\ \Delta S_2;\ i? : T \mid preS_1 \land preS_2 \land CS_1 \land CS_2]$$
$$\sqsubseteq_{\mathcal{A}}$$
$$[\Delta S_1;\ \Xi S_2;\ i? : T \mid preS_1 \land CS_1];\ [\Xi S_1;\ \Delta S_2;\ i? : T \mid preS_2 \land CS_2]$$

provided

- $\alpha(S_1) \cap \alpha(S_2) = \varnothing$
- $FV(preS_1) \subseteq \alpha(S_1) \cup \{i?\}$
- $FV(preS_2) \subseteq \alpha(S_2) \cup \{i?\}$
- $DFV(CS_1) \subseteq \alpha(S_1')$
- $DFV(CS_2) \subseteq \alpha(S_2')$
- $UDFV(CS_2) \cap DFV(CS_1) = \varnothing$ ❑

Law C.72 (Initialisation schema/Sequence—introduction*)

$$[\, S_1';\ S_2' \mid CS_1 \land CS_2 \,]$$
$$=$$
$$[\, S_1' \mid CS_1 \,];\ [\, S_2' \mid CS_2 \,]$$

provided

- $\alpha(S_1) \cap \alpha(S_2) = \varnothing$
- $DFV(CS_1) \subseteq \alpha(S_1')$
- $DFV(CS_2) \subseteq \alpha(S_2')$ ❑

Law C.73 (Schemas/Parallelism composition—distribution*)

$$SExp;\ (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)$$
$$=$$
$$(SExp;\ A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2$$

provided

- $wrtV(SExp) \subseteq ns_1$
- $wrtV(SExp) \cap usedV(A_2) = \varnothing$ ❑

Law C.74 (Schemas/Interleaving—distribution*)

$$
\begin{array}{l}
(\Box\, i \bullet g_i \,\&\, SExp_i);\ (A_1 \,|\!|\![ns_1 \mid ns_2]\!|\!|\, A_2) \\
= \\
((\Box\, i \bullet g_i \,\&\, SExp_i);\ A_1) \,|\!|\![ns_1 \mid ns_2]\!|\!|\, A_2
\end{array}
$$

provided

- ➭ $\bigcup_i wrtV(SExp_i) \subseteq ns_1$
- ➭ $\bigcup_i wrtV(SExp_i) \cap usedV(A_2) = \varnothing$ ❑

Law C.75 (Schemas refinement*)

$$SExp_1 \sqsubseteq_{\mathcal{A}} SExp_2$$

where

- $SExp_1 \mathrel{\widehat=} [\Delta S;\ di?;\ do! \mid P_1]$
- $SExp_2 \mathrel{\widehat=} [\Delta S;\ di?;\ do! \mid P_2]$

provided

- ➭ pre $SExp_1 \Rightarrow$ pre $SExp_2$
- ➭ (pre $SExp_1 \wedge P_2) \Rightarrow P_1$ ❑

Parallelism composition

Law C.76 (Parallelism composition commutativity*)

$$A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 = A_2 \,[\![\, ns_2 \mid cs \mid ns_1 \,]\!]\, A_1$$

Law C.77 (Partition expansion*)

$$
\begin{array}{l}
\textbf{var}\ x : T \bullet A_1;\ (A_2 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_3) \\
= \\
\textbf{var}\ x : T \bullet A_1;\ (A_2 \,[\![\, ns_1 \cup \{x\} \mid cs \mid ns_2 \,]\!]\, A_3)
\end{array}
$$

provided $x \notin ns_2$ ❑

Law C.78 (Parallelism composition introduction 1*)

$$c \to A = (c \to A \llbracket ns_1 \mid \{\!| c |\!\} \mid ns_2 \rrbracket c \to Skip)$$
$$c.e \to A = (c.e \to A \llbracket ns_1 \mid \{\!| c |\!\} \mid ns_2 \rrbracket c.e \to Skip)$$

provided

- $c \notin usedC(A)$
- $wrtV(A) \subseteq ns_1$ ❑

Law C.79 (Sequence/Parallelism composition—introduction 1)

$$A_1;\ A_2(e)$$
$$=$$
$$((A_1;\ c!e \to Skip) \llbracket \overline{wrtV(A_2)} \mid \{\!|c|\!\} \mid wrtV(A_2) \rrbracket c?y \to A_2(y)) \setminus \{\!|c|\!\}$$

provided

- $c \notin usedC(A_1) \cup usedC(A_2)$
- $y \notin FV(A_2)$
- $wrtV(A_1) \cap usedV(A_2) = \varnothing$
- $FV(e) \cap wrtV(A_2\ before\ e) = \varnothing$ ❑

Law C.80 (Channel extension 1)

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_1 \llbracket ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \rrbracket A_2$$

provided $c \notin usedC(A_1) \cup usedC(A_2)$ ❑

Law C.81 (Channel extension 2)

$$A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2(e)$$
$$=$$
$$(c!e \to A_1 \llbracket ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \rrbracket c?x \ \to A_2(x)) \setminus \{\!|c|\!\}$$

provided

- $c \notin usedC(A_1) \cup usedC(A_2)$
- $x \notin FV(A_2)$
- $FV(e) \cap wrtV(A_2\ before\ e) = \varnothing$ ❑

Law C.82 (Channel extension 3*)

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2(e)) \setminus cs_2$$
$$=$$
$$((c!e \to A_1) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (c?x \to A_2(x))) \setminus cs_2$$

provided

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

Law C.83 (Channel extension 4*)

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = ((c \to A_1) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (c \to A_2)) \setminus cs_2$$
$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = ((c.e \to A_1) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (c.e \to A_2)) \setminus cs_2$$

provided

- ➩ $c \in cs_1$
- ➩ $c \in cs_2$ ❑

Law C.84 (Parallelism composition/Sequence—step*)

$$(A_1;\ A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3 = A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)$$

provided

- ➩ $initials(A_3) \subseteq cs$
- ➩ $cs \cap usedC(A_1) = \varnothing$
- ➩ $wrtV(A_1) \cap usedV(A_3) = \varnothing$
- ➩ A_3 *is divergence-free*
- ➩ $wrtV(A_1) \subseteq ns_1$ ❑

Law C.85 (Parallelism composition/External choice—exchange)

$$(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) \Box (B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2)$$
$$=$$
$$(A_1 \Box B_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_2 \Box B_2)$$

provided $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2 = A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_1 = Stop$ ❑

Law C.86 (Parallelism composition/External choice—expansion*)

$$(\Box\, i \bullet a_i \to A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\Box\, j \bullet b_j \to B_j)$$
$$=$$
$$(\Box\, i \bullet a_i \to A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket ((\Box\, j \bullet b_j \to B_j) \Box (c \to C))$$

provided

- $\bigcup_i \{a_i\} \subseteq cs$
- $c \in cs$
- $c \notin \bigcup_i \{a_i\}$
- $c \notin \bigcup_j \{b_j\}$

Law C.87 (Parallelism composition/External choice—distribution*)

$$\Box\, i \bullet (A_i \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A) = (\Box\, i \bullet A_i) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A$$

provided

➩ $initials(A) \subseteq cs$

➩ A *is deterministic* ❑

Law C.88 (Parallelism composition/Sequence—distribution*)

$$(A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2);\, (B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2)$$
$$=$$
$$(A_1;\, B_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_2;\, B_2)$$

provided

➩ $initials(B_1) \cup initials(B_2) \subseteq cs$

➩ $usedC(A_1) \cap initials(B_2) = \varnothing$

➩ $usedC(A_2) \cap initials(B_1) = \varnothing$

➩ $usedV(B_1) \cap ns_2 = usedV(B_2) \cap ns_1 = \varnothing$ ❑

Law C.89 (Parallelism composition Assignment/*Skip**)

$$vl := el \llbracket ns_1 \mid cs \mid ns_2 \rrbracket Skip = vl := el$$

provided

➩ ns_1 *and* ns_2 *partition the variables in scope*

➩ $vl \in ns_1$ ❑

Law C.90 (Parallelism composition unit*)

$$Skip \llbracket ns_1 \mid cs \mid ns_2 \rrbracket Skip = Skip$$

Law C.91 (Parallelism composition unit 2*)

$$Stop \llbracket ns_1 \mid cs \mid ns_2 \rrbracket Stop = Stop$$

Law C.92 (Parallelism composition Deadlocked 1*)

$$(c_1 \to A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \to A_2) = Stop = Stop \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (c_2 \to A_2)$$

provided

- $c_1 \neq c_2$
- $\{c_1, c_2\} \subseteq cs$ ❑

Law C.93 (Parallelism composition Deadlocked 2)

$$g1 \;\&\; c_1 \to A_1 \llbracket ns_1 \mid cs \cup \{\!| c_1, c_2 |\!\} \mid ns_2 \rrbracket g_2 \;\&\; c_2 \to A_2 = Stop$$

provided

- $c_1 \neq c_2$
- $\{c_1, c_2\} \subseteq cs$ ❑

Law C.94 (Parallelism composition Zero*)

$$Chaos \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A = Chaos$$

Interleaving

Law C.95 (Interleaving/Sequence—distribution*)

$$(A_1 \;|\!|\!\![ns_1 \mid ns_2]\!\!|\!|\; A_2); (B_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket B_2)$$
$$=$$
$$(A_1; B_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_2; B_2)$$

provided

- $(usedC(A_1) \cup usedC(A_2)) \cap cs = \varnothing$
- $initials(B_1) \cup initials(B_2) \subseteq cs$ ❑

Law C.96 (Interleaving Zero*)

$$Chaos \; [\![ns_1 \mid ns_2]\!] \; A = Chaos$$

Law C.97 (Interleaving *Stop)**

$$Stop \; [\![ns_1 \mid ns_2]\!] \; Stop = Stop$$

Law C.98 (Parallelism composition/Interleaving Equivalence*)

$$A_1 \; [\![ns_2 \mid ns_2]\!] \; A_2 = A_1 \; [\![\, ns_2 \mid \varnothing \mid ns_2 \,]\!] \; A_2$$

Law C.99 (Interleaving Choices*)

$$\begin{array}{l} (c_1 \rightarrow A_1) \; [\![ns_1 \mid ns_2]\!] \; (c_2 \rightarrow A_2) \\ = \\ c_1 \rightarrow (A_1 \; [\![ns_1 \mid ns_2]\!] \; (c_2 \rightarrow A_2)) \; \Box \; c_2 \rightarrow ((c_1 \rightarrow A_1) \; [\![ns_1 \mid ns_2]\!] \; A_2) \end{array}$$

Prefix

Law C.100 (Prefix/*Skip)**

$$c \rightarrow A = (c \rightarrow Skip);\ A$$

$$c.e \rightarrow A = (c.e \rightarrow Skip);\ A$$

Law C.101 (Prefix/Sequential composition—associativity)

$$c \rightarrow (A_1;\ A_2) = (c \rightarrow A_1);\ A_2$$

$$c.e \rightarrow (A_1;\ A_2) = (c.e \rightarrow A_1);\ A_2$$

provided $FV(A_2) \cap \alpha(c) = \varnothing$ ❑

Law C.102 (Prefix/Hiding*)

$$(c \rightarrow Skip) \setminus \{c\} = Skip$$

$$(c.e \rightarrow Skip) \setminus \{c\} = Skip$$

Law C.103 (Prefix introduction*)

$$A = (c \rightarrow A) \setminus \{\!| \; c \; |\!\}$$

provided $c \notin usedC(A)$ ❑

Law C.104 (Prefix/External choice—distribution*)

$$c \rightarrow \Box\, i \bullet g_i \,\&\, A_i = \Box\, i \bullet g_i \,\&\, c \rightarrow A_i$$

provided

- $\vee\, i \bullet g_i$
- $\forall\, i, j \mid i \neq j \bullet \neg\,(g_i \wedge g_j)$ *(guards are mutually exclusive).* ❑

Law C.105 (Prefix/Internal choice—distribution)

$$c \rightarrow (A_1 \sqcap A_2) = (c \rightarrow A_1) \sqcap (c \rightarrow A_2)$$
$$c.e \rightarrow (A_1 \sqcap A_2) = (c.e \rightarrow A_1) \sqcap (c.e \rightarrow A_2)$$

Law C.106 (Prefix/Parallelism composition—distribution)

$$c \rightarrow (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2) = (c \rightarrow A_1) \llbracket\, ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \,\rrbracket\, (c \rightarrow A_2)$$
$$c.e \rightarrow (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2) = (c.e \rightarrow A_1) \llbracket\, ns_1 \mid cs \cup \{\!|c|\!\} \mid ns_2 \,\rrbracket\, (c.e \rightarrow A_2)$$

provided $c \notin usedC(A_1) \cup usedC(A_2)$ *or* $c \in cs$ ❑

Law C.107 (Communication/Parallelism composition—distribution)

$$(c!e \rightarrow A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, (c?x \rightarrow A_2(x)) = c.e \rightarrow (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2(e))$$

provided

- $c \in cs$
- $x \notin FV(A_2)$. ❑

Law C.108 (Input prefix/Parallelism composition—distribution*)

$$c?x \rightarrow (A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2) = (c?x \rightarrow A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, (c?x \rightarrow A_2)$$

provided

$c \in cs$ ❑

Law C.109 (Input prefix/Parallelism composition—distribution 2*)

$$c?x \to (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2) = (c?x \to A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$$

provided

- $c \notin cs$
- $x \notin usedV(A_2)$
- $initials(A_2) \subseteq cs$
- A_2 *is deterministic* ❑

External choice

Law C.110 (External choice commutativity*)

$$A_1 \square A_2 = A_2 \square A_1$$

Law C.111 (External choice elimination*)

$$A \square A = A$$

Law C.112 (External choice/Sequence—distribution)

$$(\square\, i \bullet g_i \,\&\, c_i \to A_i);\ B = \square\, i \bullet g_i \,\&\, c_i \to A_i;\ B$$

Law C.113 (External choice/Sequence—distribution 2*)

$$((g_1 \,\&\, A_1) \square (g_2 \,\&\, A_2));\ B = ((g_1 \,\&\, A_1);\ B) \square ((g_2 \,\&\, A_2);\ B)$$

provided $g_1 \Rightarrow \neg\, g_2$ ❑

Law C.114 (External choice unit)

$$Stop \square A = A$$

Internal Choice

Law C.115 (Sequence/Internal choice—distribution*)

$$A_1;\ (A_2 \sqcap A_3) = (A_1;\ A_2) \sqcap (A_1;\ A_3)$$

Law C.116 (Internal choice elimination*)

$$A \sqcap A = A$$

Law C.117 (Internal choice elimination 2*)

$$A_1 \sqcap A_2 \sqsubseteq_{\mathcal{A}} A_1$$

Law C.118 (Internal choice zero*)

$$A \sqcap Chaos = Chaos$$

Law C.119 (Internal choice/Parallelism composition Distribution*)

$$\begin{array}{l} (A_1 \sqcap A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3 \\ = \\ (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3) \sqcap (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3) \end{array}$$

Hiding

Law C.120 (Hiding Identity*)

$$A \setminus cs = A$$

provided $cs \cap usedC(A) = \varnothing$ ❑

Law C.121 (Hiding combination)

$$(A \setminus cs_1) \setminus cs_2 = A \setminus (cs_1 \cup cs_2)$$

Law C.122 (Hiding/External choice—distribution*)

$$(A_1 \Box A_2) \setminus cs = (A_1 \setminus cs) \Box (A_2 \setminus cs)$$

provided $(initials(A_1) \cup initials(A_2)) \cap cs = \varnothing$ ❑

Law C.123 (Hiding/External choice—distribution 2*)

$$((g_1 \,\&\, A_1) \Box (g_2 \,\&\, A_2)) \setminus cs = ((g_1 \,\&\, A_1) \setminus cs) \Box ((g_2 \,\&\, A_2) \setminus cs)$$

provided $\neg\,(g_1 \wedge g_2)$ *or* $(initials(A_1) \cup initials(A_2)) \cap cs = \varnothing$ ❑

Law C.124 (Hiding expansion 2*)

$$A \setminus cs = A \setminus cs \cup \{c\}$$

provided $c \notin usedC(A)$ ❑

Law C.125 (Hiding/Sequence—distribution*)

$$(A_1;\ A_2) \setminus cs = (A_1 \setminus cs);\ (A_2 \setminus cs)$$

Law C.126 (Hiding/*Chaos*—distribution*)

$$Chaos \setminus cs = Chaos$$

Law C.127 (Hiding/Parallelism composition—distribution*)

$$(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = (A_1 \setminus cs_2) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (A_2 \setminus cs_2)$$

provided $cs_1 \cap cs_2 = \varnothing$ ❑

Recursion

Law C.128 (Recursion unfold)

$$\mu X \bullet F(X) = F(\mu X \bullet F(X))$$

Law C.129 (Recursion—least fixed-point)

$$F(Y) \sqsubseteq_{\mathcal{A}} Y \Rightarrow \mu X \bullet F(X) \sqsubseteq_{\mathcal{A}} Y$$

Law C.130 (Recursion Refinement*)

$$\mu X \bullet F_1(X) \sqsubseteq_{\mathcal{A}} \mu X \bullet F_2(X)$$

provided $F_1 \sqsubseteq_{\mathcal{A}} F_2$ ❑

Law C.131 (Recursion—divergence introduction*)

$$(\mu X \bullet (c \rightarrow X)) \setminus \{c\} = (\mu X \bullet (c.e \rightarrow X)) \setminus \{c\} = Chaos$$

Sequence

Law C.132 (Sequence unit)

$$(A) Skip;\ A$$
$$(B) A = A;\ Skip$$

Law C.133 (Sequence zero)

$$Stop;\ A = Stop$$

Law C.134 (Sequence zero 2*)

$$Chaos;\ A = Chaos$$

Chaos

Law C.135 ($Chaos$ Refinement*)

$$Chaos \sqsubseteq_{\mathcal{A}} A$$

Variable Blocks

Law C.136 (Variable block introduction*)

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

provided $x \notin FV(A)$ ❑

Law C.137 (Variable block/Sequence—extension*)

$$A_1;\ (\mathbf{var}\ x : T \bullet A_2);\ A_3 = (\mathbf{var}\ x : T \bullet A_1;\ A_2;\ A_3)$$

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

Law C.138 (Variable block/Parallelism composition—extension*)

$$\begin{array}{l} (\mathbf{var}\ x : T \bullet A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 \\ = \\ (\mathbf{var}\ x : T \bullet A_1 \llbracket ns_1 \cup \{x\} \mid cs \mid ns_2 \rrbracket A_2) \end{array}$$

provided $x \notin FV(A_2) \cup ns_1 \cup ns_2$ ❑

Law C.139 (Variable Substitution*)

$$A(x) = \mathbf{var}\ y \bullet y : [y' = x];\ A(y)$$

provided y *is not free in* A ❑

Alternation

Law C.140 (Alternation Introduction*)

$$w : [pre, post] \sqsubseteq_{\mathcal{A}} \textbf{if } \|_i g_i \rightarrow\ w : [g_i \wedge pre, post] \textbf{ fi}$$

provided $pre \Rightarrow \bigvee_i g_i$ ❑

Law C.141 (Alternation/Guarded Actions—interchange*)

$$\textbf{if } g_1 \rightarrow\ A_1 \| \ g_2 \rightarrow A_2 \textbf{ fi} = g_1 \ \&\ A_1 \ \square\ g_2 \ \&\ A_2$$

provided

➩ $g_1 \vee g_2$
➩ $g_1 \Rightarrow \neg\, g_2$ ❑

Substitution

Law C.142 (Substitution introduction*)

$$A = A[old_1, ..., old_n := new_1, ..., new_n]$$

provided $\{old_1, ..., old_n\} \cap FV(A) = \varnothing$ ❑

Law C.143 (Substitution expansion*)

$$F(A[old_1, \ldots, old_n := new_1, \ldots, new_n]) = F(A)[old_1, \ldots, old_n := new_1, \ldots, new_n]$$

provided $\{old_1, \ldots, old_n\} \cap FV(F(\_)) = \varnothing$ ❑

Law C.144 (Substitution combination*)

$$A[old_1, \ldots, old_n := mid_1, \ldots, mid_n][mid_1, \ldots, mid_n := new_1, \ldots, new_n]$$
$$=$$
$$A[old_1, \ldots, old_n := new_1, \ldots, new_n]$$

provided $\{mid_1, \ldots, mid_n\} \cap FV(A) = \varnothing$ ❑

Law C.145 (Substitution combination 2*)

$$A[old_1, ..., old_n := new_1, \ldots, new_n][old_{n+1}, ..., old_m := new_{n+1}, ..., new_m]$$
$$=$$
$$A[old_1, ..., old_m := new_1, ..., new_m]$$

provided $\{new_1, ..., new_n\} \cap \{old_{n+1}, ..., old_m\} = \varnothing$ ❑

Process Refinement

Law C.146 (Process splitting)

Let qd and rd stand for the declarations of the processes Q and R, determined by Q.State, Q.PPar, and Q.Act, and R.State, R.PPar, and R.Act, respectively, and pd stand for the process declaration.

$$\begin{array}{l} \textbf{process } P \mathrel{\widehat=} \textbf{begin state } State \mathrel{\widehat=} Q.State \wedge R.State \\ \qquad\qquad\qquad\qquad Q.PPar \wedge_{\Xi} R.State \\ \qquad\qquad\qquad\qquad R.PPar \wedge_{\Xi} Q.State \\ \qquad\qquad\qquad\qquad \bullet\ F(Q.Act, R.Act) \\ \qquad\qquad\quad \textbf{end} \end{array}$$

Then

$$pd = (qd\ rd\ \textbf{process } P \mathrel{\widehat=} F(Q, R))$$

provided *Q.PPar and R.PPar are disjoint with respect to R.State and Q.State.* ❑

Law C.147 (Process Splitting 2*)

$$
\begin{array}{l}
\textbf{process } G \mathrel{\widehat=} \textbf{begin} \\
\quad LState \mathrel{\widehat=} [\, id : Range;\ comps \mid pred_l \,] \\
\quad \textbf{state } GState \mathrel{\widehat=} \\
\qquad\qquad [\, f : Range \to LState \mid \forall\, j : Range \bullet (f\ j).id = j \land pred_g(j) \,] \\
\quad L.schema_j \land_{\Xi}\ GState \\
\quad L.action_k \land_{\Xi}\ GState
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Promotion} \\
\hline
\Delta LState;\ \Delta GState;\ id? : Range \\
\hline
\theta LState = f\ \ id? \land f' = f \oplus \{id? \mapsto \theta LState'\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\quad G.schema_j \mathrel{\widehat=} \forall\, id? : Range \bullet L.schema_j \land Promotion \\
\quad G.action_k \mathrel{\widehat=} \\
\qquad \lbrack\!\lbrack\, cs \,\rbrack\!\rbrack\ i : Range \bullet \lbrack\!\lbrack\, \alpha\,(f\ i) \,\rbrack\!\rbrack \bullet (\mathbf{promote_2}\ L.action_k)\,[id, id? := i, i] \\
\quad \bullet\ G.action\ \textbf{end} \\
= \\
\textbf{process } L \mathrel{\widehat=}\ (id : Range \bullet \textbf{begin state } LState \mathrel{\widehat=} [\, comps \mid pred_l \,] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad L.schema_j\ \ L.action_k \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \bullet\ L.action\ \textbf{end}) \\
\textbf{process } G \mathrel{\widehat=}\ \lbrack\!\lbrack cs \rbrack\!\rbrack\ id : Range \bullet L(id)
\end{array}
$$

Appendix D

Refinement of Mutually Recursive Actions

In this appendix we present the motivation for the syntactic sugar for mutually recursive actions, used to improve the presentation of refinements and processes. The proof of the theorem used in the refinement of such actions is also presented here.

A Simple Example Consider the following mutually recursive action definitions S, S_l, and S_r.

$$\left\{\begin{array}{l} S = \mu\, X, Y \bullet F(X, Y) \\ F(X, Y) = [a \to SExp_1; X \,\square\, b \to Y, c \to SExp_2; Y \,\square\, d \to X] \end{array}\right.$$

$$\left\{\begin{array}{l} S_l = \mu\, X, Y \bullet F_l(X, Y) \\ F_l(X, Y) = [a \to X \,\square\, b \to Y, c \to Y \,\square\, d \to X] \end{array}\right.$$

$$\left\{\begin{array}{l} S_r = \mu\, X, Y \bullet F_r(X, Y) \\ F_r(X, Y) = [a \to SExp_1; X \,\square\, b \to Y, c \to SExp_2; Y \,\square\, d \to X] \end{array}\right.$$

Now, suppose we want to prove that

$$S \sqsubseteq_{\mathcal{V}} [(S_l.1 \parallel S_r.1), (S_l.2 \parallel S_r.2)]$$

In order to illustrate the motivation for simplifying the notation of vectorial refinement, we present the proof of the vectorial refinement presented above. First, we apply the definition of S, S_l and S_r.

$$\begin{array}{l} S \sqsubseteq_{\mathcal{V}} [(S_l.1 \parallel S_r.1), (S_l.2 \parallel S_r.2)] \\ \widehat{=}\ \ [\text{Definitions of } S, S_l, \text{ and } S_r] \\ \mu\, X, Y \bullet F(X, Y) \sqsubseteq_{\mathcal{V}} \left[\begin{array}{l} (\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1, \\ (\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2 \end{array}\right] \end{array}$$

Next, we may use a vectorial version of the recursion least fixed-point law.

$$
\begin{array}{l}
\Leftarrow [\text{Vectorial version of law C.129 (recursion least fixed-point)}] \\
F\left(\begin{array}{l}(\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1, \\ (\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2\end{array}\right) \\
\sqsubseteq_{\mathcal{V}} \\
\left[\begin{array}{l}(\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1, \\ (\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2\end{array}\right]
\end{array}
$$

So, we conclude that, to prove the initial refinement, we can prove the refinement above. We start the proof of this refinement by applying the definition of F.

$$
\begin{array}{l}
F\left(\begin{array}{l}(\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1, \\ (\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2\end{array}\right) \\
\widehat{=}\ [\text{Definition of } F] \\
\left[\begin{array}{l}
\left(\begin{array}{l} a \rightarrow SExp_1;((\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1) \\ \Box\ b \rightarrow ((\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2)\end{array}\right), \\
\left(\begin{array}{l} c \rightarrow SExp_2;((\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2) \\ \Box\ d \rightarrow ((\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1)\end{array}\right)
\end{array}\right]
\end{array}
$$

Next, we distribute the schema over the parallelism as follows.

$$
\begin{array}{l}
= [C.76, C.73] \\
\left[\begin{array}{l}
\left(\begin{array}{l} a \rightarrow ((\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (SExp_1;(\mu\, X, Y \bullet F_r(X, Y)).1)) \\ \Box\ b \rightarrow ((\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (\mu\, X, Y \bullet F_r(X, Y)).2)\end{array}\right), \\
\left(\begin{array}{l} c \rightarrow ((\mu\, X, Y \bullet F_l(X, Y)).2 \parallel (SExp_2;(\mu\, X, Y \bullet F_r(X, Y)).2)) \\ \Box\ d \rightarrow ((\mu\, X, Y \bullet F_l(X, Y)).1 \parallel (\mu\, X, Y \bullet F_r(X, Y)).1)\end{array}\right)
\end{array}\right]
\end{array}
$$

Then, as the channels a, b, c, and d are in the synchronisation channel set, we may apply the distribution of prefix over parallelism law.

$$
\begin{array}{l}
= [C.106] \\
\left[\begin{array}{l}
\left(\begin{array}{l} (a \rightarrow (\mu\, X, Y \bullet F_l(X, Y)).1) \parallel (a \rightarrow SExp_1;(\mu\, X, Y \bullet F_r(X, Y)).1) \\ \Box\ (b \rightarrow (\mu\, X, Y \bullet F_l(X, Y)).2) \parallel (b \rightarrow (\mu\, X, Y \bullet F_r(X, Y)).2)\end{array}\right), \\
\left(\begin{array}{l} (c \rightarrow (\mu\, X, Y \bullet F_l(X, Y)).2) \parallel (c \rightarrow SExp_2;(\mu\, X, Y \bullet F_r(X, Y)).2) \\ \Box\ (d \rightarrow (\mu\, X, Y \bullet F_l(X, Y)).1) \parallel (d \rightarrow (\mu\, X, Y \bullet F_r(X, Y)).1)\end{array}\right)
\end{array}\right]
\end{array}
$$

Next, we apply the exchange of parallelism and external choice law. This application is valid since the initials of all actions are in the synchronisation channel set.

$$
\begin{array}{l}
= [C.85] \\
\left[\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} a \to (\mu X, Y \bullet F_l(X, Y)).1 \\ \Box\, b \to (\mu X, Y \bullet F_l(X, Y)).2 \end{array} \right) \\
\parallel \\
\left(\begin{array}{l} a \to SExp_1;(\mu X, Y \bullet F_r(X, Y)).1 \\ \Box\, b \to (\mu X, Y \bullet F_r(X, Y)).2 \end{array} \right)
\end{array} \right), \\
\left(\begin{array}{l}
\left(\begin{array}{l} c \to (\mu X, Y \bullet F_l(X, Y)).2 \\ \Box\, d \to (\mu X, Y \bullet F_l(X, Y)).1) \end{array} \right) \\
\parallel \\
\left(\begin{array}{l} c \to SExp_2;(\mu X, Y \bullet F_r(X, Y)).2 \\ \Box\, d \to (\mu X, Y \bullet F_r(X, Y)).1 \end{array} \right)
\end{array} \right)
\end{array} \right]
\end{array}
$$

The definition of array allows us to extend the action above as follows.

$$
\begin{array}{l}
\widehat{=} [A = [A, B].1, B = [A, B].2] \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left[\begin{array}{l}
\left(\begin{array}{l} a \to (\mu X, Y \bullet F_l(X, Y)).1 \\ \Box\, b \to (\mu X, Y \bullet F_l(X, Y)).2 \end{array} \right), \\
\left(\begin{array}{l} c \to (\mu X, Y \bullet F_l(X, Y)).2 \\ \Box\, d \to (\mu X, Y \bullet F_l(X, Y)).1 \end{array} \right)
\end{array} \right].1 \\
\parallel \\
\left[\begin{array}{l}
\left(\begin{array}{l} a \to SExp_1;(\mu X, Y \bullet F_r(X, Y)).1 \\ \Box\, b \to (\mu X, Y \bullet F_r(X, Y)).2 \end{array} \right), \\
\left(\begin{array}{l} c \to SExp_2;(\mu X, Y \bullet F_r(X, Y)).2 \\ \Box\, d \to (\mu X, Y \bullet F_r(X, Y)).1 \end{array} \right)
\end{array} \right].1
\end{array} \right], \\
\left[\begin{array}{l}
\left[\begin{array}{l}
\left(\begin{array}{l} a \to (\mu X, Y \bullet F_l(X, Y)).1 \\ \Box\, b \to (\mu X, Y \bullet F_l(X, Y)).2 \end{array} \right), \\
\left(\begin{array}{l} c \to (\mu X, Y \bullet F_l(X, Y)).2 \\ \Box\, d \to (\mu X, Y \bullet F_l(X, Y)).1 \end{array} \right)
\end{array} \right].2 \\
\parallel \\
\left[\begin{array}{l}
\left(\begin{array}{l} a \to SExp_1;(\mu X, Y \bullet F_r(X, Y)).1 \\ \Box\, b \to (\mu X, Y \bullet F_r(X, Y)).2, \end{array} \right), \\
\left(\begin{array}{l} c \to SExp_2;(\mu X, Y \bullet F_r(X, Y)).2 \\ \Box\, d \to (\mu X, Y \bullet F_r(X, Y)).1 \end{array} \right)
\end{array} \right].2
\end{array} \right]
\end{array} \right]
\end{array}
$$

Finally, using a vectorial version of the recursion unfolding law, we conclude our proof.

$$
\begin{array}{l}
= [\text{Vectorial version of law C.128 (Recursion Unfolding)}] \\
\left[\begin{array}{l}
(\mu X, Y \bullet F_l(X, Y)).1 \parallel (\mu X, Y \bullet F_r(X, Y)).1, \\
(\mu X, Y \bullet F_l(X, Y)).2 \parallel (\mu X, Y \bullet F_r(X, Y)).2
\end{array} \right]
\end{array}
$$

❑

Simplification The system and refinement that we have just proved above are quite simply presented and understood. However, it may be the case that the system has quite a complicated presentation in the above notation. Our case study is a good example of such a system. For this reason, we have adopted a notation for the presentation of all mutually recursive systems and of refinements on these systems in a more concise way.

First, let us generalise the syntactic sugar for the definitions of mutually recursive systems: every mutually recursive system of the form

$$\begin{cases} S \mathrel{\widehat{=}} \mu\, X_0, \ldots, X_n \bullet F(X_0, \ldots, X_n) \\ F(X_0, \ldots, X_n) \mathrel{\widehat{=}} [F_0(X_0, \ldots, X_n), \ldots, F_n(X_0, \ldots, X_n)] \end{cases}$$

can be presented using the following syntax.

$$S_S \mathrel{\widehat{=}} [N_0, \ldots, N_n]$$

For each index i in $0\,..\,n$, the action N_i is defined as $N_i \mathrel{\widehat{=}} G_i$, where G_i is defined as its corresponding $F_i(X_0, \ldots, X_n)$, but replacing all the occurrences of the variables $X_0, \ldots, X_n$ by the corresponding $N_0, \ldots, N_n$. Furthermore, the names N_i are fresh.

$$G_i = F_i[N_0, \ldots, N_n / X_0, \ldots, X_n]$$

In our example, we use this to get the following syntactic sugaring S_S of the process S.

$$S_S \mathrel{\widehat{=}} [N_0, N_1]$$

$$N_0 \mathrel{\widehat{=}} G_0 \ \textit{where}\ G_0 = a \rightarrow \mathit{SExp}_1; N_0 \;\Box\; b \rightarrow N_1$$
$$N_1 \mathrel{\widehat{=}} G_1 \ \textit{where}\ G_1 = c \rightarrow \mathit{SExp}_2; N_1 \;\Box\; d \rightarrow N_0$$

We may also apply the strategy to get the syntactic sugaring S_{l_S} of the process S_l.

$$S_{l_S} \mathrel{\widehat{=}} [N_{l_1}, N_{l_2}]$$

$$N_{l_1} \mathrel{\widehat{=}} G_{l_1} \ \textit{where}\ G_{l_1} = a \rightarrow N_{l_1} \;\Box\; b \rightarrow N_{l_2}$$
$$N_{l_2} \mathrel{\widehat{=}} G_{l_2} \ \textit{where}\ G_{l_2} = c \rightarrow N_{l_2} \;\Box\; d \rightarrow N_{l_1}$$

In a similar way, we also apply the strategy to get the syntactic sugaring S_{r_S} of the process S_r.

$$S_{r_S} \mathrel{\widehat{=}} [N_{r_1}, N_{r_2}]$$

$$N_{r_1} \mathrel{\widehat{=}} G_{r_1} \ \textit{where}\ G_{r_1} = a \rightarrow \mathit{SExp}_1; N_{r_1} \;\Box\; b \rightarrow N_{r_2}$$
$$N_{r_2} \mathrel{\widehat{=}} G_{r_2} \ \textit{where}\ G_{r_2} = c \rightarrow \mathit{SExp}_2; N_{r_2} \;\Box\; d \rightarrow N_{r_1}$$

Now, we present the syntactic sugaring for proving refinements of these systems. We want to prove a refinement of the following form, where $Y_0, \ldots, Y_n$ are actions.

$$S \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n]$$

To prove this property, we can apply the vectorial version of the recursion-least fixed-point

Law (C.129) as follows.

[Vectorial version of law C.129 (Recursion-Least Fixed Point)]

$\Leftarrow F(Y_0, \ldots, Y_n) \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n]$

Applying the definition of F we get the following proof obligation.

$= [F_0(Y_0, \ldots, Y_n), \ldots, F_n(Y_0, \ldots, Y_n)] \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n]$ [Definition of F]

The previous proof obligation can then be transformed as follows.

$F_i(Y_0, \ldots, Y_n) \sqsubseteq_{\mathcal{A}} Y_i$
$= F_i[Y_0, \ldots, Y_n / X_0, \ldots, X_n] \sqsubseteq_{\mathcal{A}} Y_i$ [Function Invocation]
$= F_i[N_0, \ldots, N_n / X_0, \ldots, X_n][Y_0, \ldots, Y_n / N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_i$
[Renaming Composition]
$= G_i[Y_0, \ldots, Y_n / N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_i$ [Definition of G_i]

We have then the following proof obligation.

$$[G_0[Y_0, \ldots, Y_n / N_0, \ldots, N_n], \ldots, G_n[Y_0, \ldots, Y_n / N_0, \ldots, N_n]] \sqsubseteq_{\mathcal{V}} [Y_0, \ldots, Y_n]$$

Finally, by the definition of vectorial refinement (Definition 5.1), this refinement is valid if the refinement holds for each corresponding element in both vectors. This justifies our syntactic sugaring for proving refinements on mutually recursive systems, which is summarised in the theorem below.

Theorem D.1 (Refinement on Mutually Recursive Actions) *For a given vector of actions S_S defined in the form $S_S \mathrel{\widehat{=}} [N_0, \ldots, N_n]$, where $N_i \mathrel{\widehat{=}} F_i(N_0, \ldots, N_n)$:*

$$S_S \sqsubseteq_{\mathcal{A}} [Y_0, \ldots, Y_n] \Leftarrow \left(\begin{array}{l} F_0[Y_0, \ldots, Y_n / N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_0, \\ \ldots, \\ F_n[Y_0, \ldots, Y_n / N_0, \ldots, N_n] \sqsubseteq_{\mathcal{A}} Y_n \end{array} \right)$$

In order to prove that a vector of actions S_S as defined above is refined by a vector of actions $[Y_0, \ldots, Y_n]$, it is enough to show that, for each action N_i in S_S, we can prove that its definition F_i, if we replace $N_0, \ldots, N_n$ with $Y_0, \ldots, Y_n$ in F_i, is refined by Y_i. ❑

Back to the Simple Example The refinement we need to prove is the following.

$S_S \sqsubseteq_{\mathcal{V}} [(S_{l_S}.1 \parallel S_{r_S}.1), (S_{l_S}.2 \parallel S_{r_S}.2)]$

By the definition of S_{l_S} and S_{r_S}, it can be rewritten as $S_S \sqsubseteq_{\mathcal{V}} [N_{l_1} \parallel N_{r_1}, N_{l_2} \parallel N_{r_2}]$. Our refinement strategy, however, gives us the following proving obligations for this refinement.

$S_S \sqsubseteq_{\mathcal{V}} [N_{l_1} \parallel N_{r_1}, N_{l_2} \parallel N_{r_2}]$
$\Leftarrow$ [Theorem D.1]
[1] $G_0[N_{l_1} \parallel N_{r_1}, N_{l_2} \parallel N_{r_2} / N_0, N_1] \sqsubseteq_{\mathcal{A}} N_{l_1} \parallel N_{r_1}$
and
[2] $G_1[N_{l_1} \parallel N_{r_1}, N_{l_2} \parallel N_{r_2} / N_0, N_1] \sqsubseteq_{\mathcal{A}} N_{l_2} \parallel N_{r_2}$

These proofs, however, can now be proved separately. We prove the refinement of G_0.

This proof starts by applying the definition of G_0 and the substitution.

[1]
$G_0[N_{l_1} \parallel N_{r_1}, N_{l_2} \parallel N_{r_2} / N_0, N_1] \sqsubseteq_{\mathcal{A}} N_{l_1} \parallel N_{r_1}$
$=$ [Definition of G_0]
$(a \rightarrow SExp_1;N_0 \mathrel{\Box} b \rightarrow N_1)[N_{l_1} \parallel N_{r_1}, N_{l_2} \parallel N_{r_2} / N_0, N_1]$
$=$ [Definition of Substitution]
$a \rightarrow SExp_1;(N_{l_1} \parallel N_{r_1}) \mathrel{\Box} b \rightarrow (N_{l_2} \parallel N_{r_2})$

Next, we distribute the schemas over the parallelism as follows.

$= [C.73]$
$a \rightarrow (N_{l_1} \parallel (SExp_1;N_{r_1})) \mathrel{\Box} b \rightarrow (N_{l_2} \parallel N_{r_2})$

Then, as channels a and b are in the synchronisation channel set, we may apply the distribution of prefix over parallelism law.

$= [C.106]$
$((a \rightarrow N_{l_1}) \parallel (a \rightarrow SExp_1;N_{r_1})) \mathrel{\Box} ((b \rightarrow N_{l_2}) \parallel (b \rightarrow N_{r_2}))$

Since the initial events of all the actions involved are in the synchronisation channel set, we may apply the exchange of parallelism and external choice law.

$= [C.85]$
$(a \rightarrow N_{l_1} \mathrel{\Box} b \rightarrow N_{l_2}) \parallel (a \rightarrow SExp_1;N_{r_1} \mathrel{\Box} b \rightarrow N_{r_2})$

By definition, we conclude our proof.

$=$ [Definition of N_{l_1} and N_{r_1}]
$N_{l_1} \parallel N_{r_1}$

The second proof obligation ([2]) can be proved in a very similar way. ❑

Glossary

Bibliography

[1] ProofPower. At http://www.lemma-one.com/ProofPower/index/index.html.

[2] *PVS*. At http://pvs.csl.sri.com/index.shtml.

[3] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] J.-R. Abrial. B#: toward a synthesis between Z and B. In D. Bert, J. P. Bowen, S. King, and M. Waldén, editors, *ZB*, volume **3582** of *LNCS*, pages 168–177. Springer-Verlag, 2003.

[5] D. Atiya. *Verification of Concurrent Safety-critical Systems: The Compliance Notation Approach*. PhD thesis, University of York, Department of Computer Science, York, 2005.

[6] D. Atiya, S. King, and J. C. P. Woodcock. A *Circus* semantics for Ravenscar protected objects. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FM 2003: 12th international FME Symposium*, volume **2805** of *LNCS*, Pisa, 2003. Springer-Verlag.

[7] B-Core. *B-Core's Website*, 2002. http://www.b-core.com.

[8] R. J. R. Back. *On The Correctness of Refinement Steps in Program Development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.

[9] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, **25**:593–624, 1988.

[10] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *PODC '83: second annual ACM symposium on principles of distributed computing*, pages 131–142, New York, USA, 1983. ACM Press.

[11] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, **13**(2–3):133–180, 1990.

[12] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software: 8th International SPIN Workshop*, pages 103–122, Toronto, Canada, 2001.

[13] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[14] R. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall, 1997.

[15] P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M.L. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, **52**:53–100, 2004.

[16] P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A refinement algebra for object-oriented programming. In L. Cardelli, editor, *ECOOP 2003: European Conference on Object-oriented Programming 2003*, volume **2743** of *LNCS*, pages 457–482. Springer-Verlag, 2003.

[17] E. Börger and R. F. Stärk. *Abstract State Machines–A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[18] M. J. Butler. An approach to the design of distributed systems with B AMN. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 223–241, London, 1997. Springer-Verlag.

[19] M. J. Butler. csp2B: a practical approach to combining CSP and B. *Formal Aspects of Computing*, **12**(3):182–198, 2000.

[20] M. J. Butler and M. Waldén. Distributed System Development in B. Technical Report TUCS-TR-53, 1996.

[21] D. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: an object-oriented extension to Z. *Formal Description Techniques, II (FORTE'89)*, pages 281–296, 1990.

[22] A. L. C. Cavalcanti. *A Refinement Calculus for Z.* PhD thesis, Oxford University Computing Laboratory, Oxford, 1997. Technical Monograph TM-PRG-123, ISBN 00902928-97-X.

[23] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. Control law diagrams in *Circus*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods Symposium*, volume **3582** of *LNCS*, pages 253–268. Springer-Verlag, 2005.

[24] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. Unifying classes and processes. *Journal of Software and Systems Modeling*, **4**(3):277–296, 2005.

[25] A. L. C. Cavalcanti and A. C. A. Sampaio. From CSP-OZ to Java with processes. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001. Submitted.

[26] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of actions in *Circus*. In *Proceedings of REFINE'2002*, Electronic Notes in Theoretical Computer Science, 2002. Invited Paper.

[27] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.

[28] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, **4**(3):277–296, 2005.

[29] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A refinement calculus for Z. *Formal Aspects of Computing*, **10**(3):267–289, 1999.

[30] A. L. C. Cavalcanti and J. C. P. Woodcock. A weakest precondition semantics for *Circus*. In *Proceedings of Communicating Processing Architectures 2002*. Concurrent Systems Engineering, IOS Press, 2002.

[31] A. L. C. Cavalcanti and J. C. P. Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In *Proceedings of the Pernambuco Summer School on Software Engineering: Refinement 2004*, 2004.

[32] A. L. C. Cavalcanti and J. C. P. Woodcock. Angelic nondeterminism and Unifying Theories of Programming. Technical Report 13-04, Computing Laboratory, University of Kent, June 2004.

[33] A. L. C. Cavalcanti and J. C. P. Woodcock. Angelic nondeterminism and Unifying Theories of Programming . In J. Derrick and E. Boiten, editors, *REFINE 2005*, volume **137** of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[34] J. Davies and S. Schneider. A Brief History of Timed CSP. In *MFPS '92: Selected papers of the meeting on Mathematical foundations of programming semantics*, pages 243–271, Amsterdam, 1995. Elsevier Science Publishers B. V.

[35] J. Derrick and G. Smith. Structural refinement in Object-Z/CSP. In W. Grieskamp, T. Stanten, and B. Stoddart, editors, *Integrated Formal Methods (IFM 2000)*, volume **1945** of *LNCS*, pages 194–213. Springer, November 2000.

[36] E. W. Dijkstra. *Notes on Structured Programming*, chapter 1, pages 1–82. Academic Press, 1972.

[37] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[38] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[39] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, volume **2**, pages 423–438. Chapman & Hall, 1997.

[40] C. Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM '98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.

[41] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik, Universität Oldenburg, 2000.

[42] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*, volume **19** of *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[43] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.82*, 2005.

[44] A. Freitas. *From Circus to Java: Implementation and Verification of a Translation Strategy*. Master's thesis, Department of Computer Science, The University of York, 2005. Submitted.

[45] L. Freitas. *Model-checking Circus*. PhD thesis, Department of Computer Science, The University of York, 2005. Submitted.

[46] A. Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-perspective Systems*. PhD thesis, School of Computing and Mathematics, University of Teeside, 1996.

[47] A. Galloway and B. Stoddart. *An operational semantics for ZCCS*. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, page 272, Washington, DC, USA, 1997. IEEE Computer Society.

[48] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. volume **78** of *LNCS*. Springer-Verlag, 1979.

[49] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[50] The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.

[51] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java threads. In W. P. Andrè Bakkers, editor, *Proceedings of WoTUG-20: Parallel Programming and Java*, pages 48–76, 1997.

[52] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[53] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, **30**(8):672–686, 1987.

[54] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[55] J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes, data and time. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM 2002: Integrated Formal Methods, Third International Conference*, volume **2335** of *LNCS*, pages 245–266. Springer, May 2002.

[56] H. Jifeng, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In G. Goos and H. Hartmants, editors, *ESOP'86 European Symposium on Programming*, volume **213** of *LNCS*, pages 187–196, March 1986.

[57] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 2nd edition, 1990.

[58] G. Jones and M. Goldsmith. *Programming in occam 2*. Prentice-Hall, 1988.

[59] L. Lai and J. W. Sanders. A refinement calculus for communicating processes with state. In Gerard O'Regan and Sharon Flynn, editors, *1st Irish Workshop on Formal Methods*, Workshops in Computing. BCS, July 1997.

[60] B. P. Mahony and J. S. Dong. Blending object-Z and timed CSP: an introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[61] B. P. Mahony and J. S. Dong. Deep semantic links of TCSP and object-Z: TCOZ approach. *Formal Aspects of Computing*, **13**(2):142–160, 2002.

[62] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[63] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[64] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, **27**(6):481–503, 1990.

[65] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, **9**(3):287–306, 1987.

[66] A. C. Mota and A. C. A. Sampaio. Model-checking CSP-Z. In E. Astesiano, editor, *Proceedings of FASE'98, Held as Part of the ETAPS'98: European Joint Conference on Theory and Practice of Software*, volume **1382** of *LNCS*, pages 205–220. Springer, March 1998.

[67] G. Nuka and J. C. P. Woodcock. Mechanising the alphabetised relational calculus. In *WMF2003: 6th Braziliam Workshop on Formal Methods*, volume **95**, pages 209–225, Campina Grande, Brazil, October 2004.

[68] G. Nuka and J. C. P. Woodcock. Mechanising a unifying theory. In *First International Symposium on Unifying Theories of Programming*, LNCS. Springer-Verlag, 2006. To Appear.

[69] E. R. Olderog. Towards a design calculus for communicating programs. In J. C. M. Baeten and J. F. Groote, editor, *CONCUR'91: Proc. of the 2nd International Conference on Concurrency Theory*, pages 61–77. Springer, Berlin, Heidelberg, 1991.

[70] M. V. M. Oliveira. *ArcAngel: a Tactic Language for Refinement and its Tool Support*. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Pernambuco, Brazil, 2002. At http://www.ufpe.br/sib/.

[71] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus – Additional Material*, 2006. At http://www.cs.york.ac.uk/circus/refinement-calculus/oliveira-phd/.

[72] M. V. M. Oliveira and A. L. C. Cavalcanti. *Tactics of refinement*. In *14th Brazilian Symposium on Software Engineering*, pages 117–132, 2000.

[73] M. V. M. Oliveira and A. L. C. Cavalcanti. From *Circus* to JCSP. In J. Davies *et al.*, editor, *Sixth International Conference on Formal Engineering Methods*, volume **3308** of *LNCS*, pages 320–340. Springer-Verlag, November 2004.

[74] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, **15**(1):28–47, 2003.

[75] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining industrial scale systems in *Circus*. In Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures*, volume **62** of *Concurrent Systems Engineering Series*, pages 281–309. IOS Press, 2004.

[76] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Formal development of industrial-scale systems. *Innovations in Systems and Software Engineering—A NASA Journal*, **1**(2):125–146, 2005.

[77] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying theories in ProofPower-Z. In *First International Symposium on Unifying Theories of Programming*, LNCS. Springer-Verlag, 2006. To Appear.

[78] C. Pahl. Towards an action refinement calculus for abstract state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *International Workshop on Abstract State Machines*, pages 326–340. Springer-Verlag, March 2000.

[79] S. C. Qin, J. S. Dong, and W. N. Chin. A semantic foundation of TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume **2805** of *LNCS*, pages 321–340. Springer-Verlag, September 2003.

[80] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[81] A. W. Roscoe and C. A. R. Hoare. The Laws of Occam Programming. Technical Report PRG-53, Computing Laboratory, Oxford University, 1986.

[82] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In D. Gollmann, editor, *ESORICS 94*, volume **1214** of *LNCS*, pages 33–54. Springer-Verlag, 1994.

[83] M. Saaltink. The Z/EVES System. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume **1212** of *LNCS*, pages 72–85, Reading, April 1997. Springer-Verlag.

[84] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right*, volume **2391** of *LNCS*, pages 451–470. Springer-Verlag, 2002.

[85] S. Schneider and H. Treharne. Communicating B machines. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 416–435. Springer-Verlag, 2002.

[86] A. Sherif and H. Jifeng. Towards a time model for *Circus*. In C. George and H. Miao, editors, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002*, volume **2495** of *LNCS*, pages 613–624. Springer-Verlag, June 2002.

[87] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems specified in Object-Z and CSP. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of FME'97*, volume **1313** of *LNCS*, pages 62–81. Springer-Verlag, 1997.

[88] G. Smith. *The Object-Z specification language.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[89] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *ICFEM'97: First IEEE International Conference on Formal Engineering Methods*, pages 293–302. IEEE Computer Society, November 1997.

[90] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems—an integration of Object-Z and CSP. *Formal Methods in Systems Design*, **18**:249–284, May 2001.

[91] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.

[92] B. Stoddart. An Introduction to the Event Calculus. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation, 10th International Conference of Z Users*, volume **1212** of *LNCS*, pages 10–34, Reading, April 1997. Springer.

[93] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In M. Hinchey and Shaoying Liu, editors, *International Conference on Formal Engineering Methods*, pages 283–292. IEEE, 1997.

[94] X. Tang and J. C. P. Woodcock. Towards mobile processes in Unifying Theories. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society Press, September 2004.

[95] H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456. Springer, June 1999.

[96] H. Treharne and S. Schneider. How to drive a B machine. In *ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 188–208, London, 2000. Springer-Verlag.

[97] K. R. Wagner, M. Nielsen, K. Havelund, and C. George. The RAISE Language, Method and Tools. In *VDM '88 VDM—The Way Ahead*, pages 376–405, 1988.

[98] P. H. Welch. Process oriented design for Java: concurrency for all. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume **1**, pages 51–57. CSREA Press, June 2000.

[99] P. H. Welch, G. S. Stiles, G. H. Hilderink, and A. P. Bakkers. CSP for Java: multithreading for all. In B. M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume **57** of *Concurrent Systems Engineering Series*. IOS Press, April 1999.

[100] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, **14**(4):221–227, 1971.

[101] J. C. P. Woodcock. Using *Circus* for Safety-Critical Applications. In *VI Brazilian Workshop on Formal Methods*, pages 1–15, Campina Grande, Brazil, 12th–14st October 2003.

[102] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[103] J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD UK, July 2001.

[104] J. C. P. Woodcock and A. L. C. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*. IEEE Press, 2001.

[105] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume **2272** of *LNCS*, pages 184–203. Springer-Verlag, 2002.

[106] J. C. P. Woodcock, A. L. C. Cavalcanti, and L. Freitas. Operational semantics for model-checking *Circus*. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume **3582** of *LNCS*, pages 237–252. Springer-Verlag, 2005.

[107] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[108] J. C. P. Woodcock, J. Davies, and C. Bolton. Abstract Data Types and Processes. In A. W. Roscoe J. Davies and J. C. P. Woodcock, editors, *Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 391–405. Palgrave, 2000.

[109] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, **40**(5):269–276, 1991.

In memoriam
My dear uncle Herbert Lêda
✰ 22/07/1954 - ✞ 21/10/2005