University of York
Department of Computer Science

MSc by Research in Computer Science

From *Circus* to Java: Implementation and Verification of a Translation Strategy

by

Angela Figueiredo de Freitas

December, 2005

To mom and dad

Acknowledgements

I am very grateful to my supervisor, Ana Cavalcanti, for her support, enthusiasm, and guidance during this research. She has given me great incentive since my undergraduate course. I am very pleased to have her as teacher, supervisor, and friend.

I would also like to thank the people who gave important contributions for the development of this work. Marcel Oliveira, for answering my many questions about the translation strategy and the *Circus* refinement calculus. Leo Freitas, for the *Circus* AST, for helping me with the CZT framework, and for discussions and insights about some aspects of the design of *JCircus*. Manuela Xavier, for the *Circus* type checker, which is part of *JCircus*, and for discussions about the *Circus* type system. A lot of the improvements on the *Circus* parser are result of joint work with Leo and Manuela. I would like to thank Jim Woodcock, for his help with the multi-synchronisation protocol, and Peter Welch, for answering my questions about the JCSP library. I would also like to thank QinetiQ for the financial support that made possible the development of this research.

Thanks to mom and dad, for their love and care, and support in difficult moments. They have encouraged me a lot, while I was in England, and since I returned to Brazil to write up this thesis. Thanks also to all my friends, those that I left in Brazil, and those that I made in England, for the emotional support and the nice moments that we shared together. Thanks to God, for His love and provision. He made me able to carry out this work.

Abstract

Circus is a formal language that combines the Z and CSP notations. It takes advantage of the strengths of each of these languages: from Z, it brings the possibility of specifying systems with complex state requirements; from CSP, it brings the support for modelling communication and concurrency. The result is a notation suitable to model state-rich reactive systems. Many other combinations of a state-based formalism with a process algebra have been proposed. However, the distinguishing feature of *Circus* is that it includes a refinement calculus, to allow a concrete specification to be developed from an abstract *Circus* specification.

Recently, a strategy for transforming from a concrete *Circus* specification to a Java program has been proposed. It consists in translation rules that, applied to each *Circus* construct in a concrete specification, result in a Java program that implements the *Circus* program. As a consequence, a method for developing Java code from an abstract *Circus* specification is now available. This method consists of firstly applying refinement laws to transform an abstract specification into a concrete one, and then applying the translation strategy to Java. The resulting Java program uses the JCSP library, a Java implementation of the CSP model for concurrency and communication.

The main contribution of our work is an implementation of the translation strategy. The resulting tool is called *JCircus*. The benefit of using *JCircus* is that it saves time, human effort and prevents errors that are typical of the activity of manually writing code. Once a concrete specification is defined, a Java implementation can be obtained within minutes.

Another important contribution of our work was the revision and correction of the original translation strategy. Moreover, we have also tackled the verification of part of the translation strategy: the multi-synchronisation protocol. The JCSP library does not implement multi-synchronisation, which was achieved with a centralised solution that uses only simple synchronisations. We used *Circus* to model a special type of multi-synchronisation and the protocol that we used to implement it in Java. Then we used the *Circus* refinement calculus to prove that the models are related by refinement. A complete formalisation of the translation strategy requires work on relating the semantics of *Circus* and Java that is outside the scope of this thesis, but we have presented a viable approach for further validation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Circus [43] is a formal language that combines the Z [44] and CSP [36] notations. Z is a state-based formal language. In this paradigm, a system is seen as a collection of variables that defines the state of the system, and operations are relations that take the system from one state to another state or retrieve information from the state. The semantics of Z is based on the set theory. Other state-based languages include the B-method, Object-Z (an object-oriented extension of Z) and Abstract State Machines.

CSP belongs to another paradigm, that of process algebras, which has a different view of a system. Here, systems are regarded as processes; it permits specifying the order in which the operations will be carried out. CSP also allows to express concurrency and provides a mechanism through which these processes can interact. The mechanisms are the channels. Messages can be passed through them from one process to another one. Process algebras, however, are poor at describing data requirements. In CSP, this can be achieved by defining parameters for processes, however, this solution is limited. CCS [28] is another example of a process algebra.

With the aim of taking the best of both worlds and providing languages suitable to describe complex systems with both data and behavioural requirements, many combinations of a state-based formalism with a process algebra have been proposed [14, 9, 24]. There are combinations of Z with CCS [19], Z with CSP, and Object-Z [10] with CSP; also, combinations of B [5] and action systems, B and CSP. When proposing a combination of formalisms, one concern is how to unify the semantics of the languages.

CSP-OZ [14] is a combination of Object-Z and CSP. Each Object-Z class contains the declaration of the interface (channels) and an optional CSP process, which defines the behaviour of the object. Each data operation of the Z object is mapped to events in the CSP process. This formalism gives a failure-divergence semantics to Object-Z classes.

The combination of CSP and B-machines defined in [37] gives a CSP semantics for B machines. Therefore, CSP operators can also be applied to B machines. A particular architecture is defined, in which each B-machine interacts with only one CSP process, called its controller. Like CSP-OZ, it also associates data operations in the B machine to events in the CSP part.

The tool csp2B [9] provides a means of combining CSP-like descriptions with standard B specifications. The aim of the tool is to convert CSP-like specifications into standard machine-readable B specifications, so that they may be animated and appropriate proof obligations may be generated. The translation is justified in terms of an operational semantics.

The fundamental difference between *Circus* and these formalisms is that, in *Circus*, the Z and CSP constructs are freely mixed within the specification. This is because *Circus* is a language for refinement. It is a unified language of specification and programming. Since programs do not separate the treatment of data and concurrency, in *Circus*, there is no separation between the Z and CSP components of a specification or program.

This makes more difficult the reuse of tools for Z and CSP to analyse *Circus* specifications. Tools for *Circus* have to be built from scratch. Also, the semantics of Z and CSP are redefined in a new model. Despite all this, *Circus* is still easy to learn by people that are familiar with both Z and CSP, because the conventions used in these languages have been preserved.

The semantics of a formal language gives a non-ambiguous interpretation for each syntactic construct of that language. It is basically a function that maps the syntactic constructs into a mathematical model that describes what the construct represents. The semantics of *Circus* is based on the Unifying Theories of Programming (UTP) [22], a formalism that proposes a unifying model for several computational paradigms. In this formalism, both state and communication aspects of concurrent systems are captured by observational variables.

Another feature of *Circus* is that it includes a refinement calculus. For that, also included in *Circus* are the specification constructs usually found in refinement calculi and Dijkstra's language of guarded commands. It also includes a strategy based on rules to translate *Circus* programs to Java code [31]. The other combinations, even though they include a theory of refinement, do not allow refinement to code in a calculational fashion like that of *Circus*.

In this work, we implement the translation strategy. We present adaptations and extensions to it, and provide additional functionality. A verification of part of the translation strategy has also been carried out.

1.1 Motivation

Circus is a new language and it has attracted the interest of industry: it is currently being considered by QinetiQ for verification of control systems [11]. The *Circus* research group is working on techniques, tools and extensions of the theory. Such contributions include: the formal definition of the *Circus* type system and implementation of a type checker for *Circus* [45]; a model checker for *Circus* [18], in the style of FDR [15]; the mechanisation of the denotational semantics of *Circus* in ProofPower-Z [30], which will serve as a basis for a theorem prover for *Circus*; extensions of the language, with the inclusion of features for specifying time [38], mobility [39] and object orientation [13].

In [29], a technique for the formal development from an abstract *Circus* specification to an executable Java program is presented. Refinement laws are defined to allow derivation of concrete *Circus* programs from abstract ones; and a translation strategy is proposed to reach a Java implementation from a concrete *Circus* program. This technique has been used successfully for refinement and implementation in Java of an industrial scale system [32].

The generated Java program uses JCSP [40], a Java library that implements some CSP primitives. JCSP was conceived with the intention of making concurrent programming in Java easier and less error prone. Java provides a built-in concurrency model based on threads and monitors. This model is hard to apply safely and scales badly with complexity. On the other hand, CSP is easy to understand, intuitive, and its underlying theory makes possible the reasoning about desired properties of concurrent systems, for example, absence of deadlock and livelock. As *Circus* is an extension of CSP, it is appropriate to make use of this library to implement *Circus* specifications as well.

One evidence of the difficulties with Java multi-threading is the number of works available in the literature warning about these difficulties. The problem becomes harder because parallel programs, as opposed to sequential programs, have a non-deterministic execution: each time the program runs, the commands may be scheduled in a different way. This means that, in case a problem appears during one particular execution, repeating the execution to investigate the problem may be useless, because the order of execution of the commands might not be the same this time, and the problem may not arise. So, debugging a parallel program is a non-trivial task.

However, concurrency is such a natural concept: it just describes how the real world works. The implementation of a simple idea should not cause so many complications, and there seems to be something wrong. The implementation of the theory of CSP in Java is an attempt to rescue this simplicity. Other libraries have been implemented with this motivation. They include CTJ and Jack [17]. These libraries, and JCSP, share the common philosophy that concurrency should be simplified in order to encourage its widespread use.

CTJ is a library that has been developed in the University of Twente. It has strong similarities with JCSP: a common core in the base algorithms for CSP primitives. Actually, teams that developed each library interacted with each other and the result is that some ideas originally introduced in JCSP were later taken up by CTJ, and vice versa. However, a motivation behind the development of CTJ was to create a Java package for creating real-time and embedded software using the CSP model. JCSP does not share this concern, and the main differences between the two libraries lie on the implementations of such features.

The Jack library came after JCSP and CTJ. Its objective was to implement features that are not present in those two libraries: a solution to the backtrack problem and multi-synchronization. The backtrack problem consists of how to implement an input channel that receives a value which is subject to a constraint. If the constraint is not satisfied, the communication is not successful, and its effects must be cancelled. A protocol has to be implemented to deal with this situation, to be able to restore the state of the system previous to the communication in the case of an unsuccessful communication.

The translation strategy that is the basis to our work used the JCSP library. Jack is the result of an MSc project work, and it is not being maintained anymore. JCSP is extensively documented and it has the concern with formal verification: the formal proof of some of the implementation features, namely, the channel implementation and the alternation, has recently been carried out [41]. Since the support for real-time features is not a concern to us, but the development of safety-critical systems is, JCSP is an obvious choice.

The soundness of the refinement rules proposed in [29] is proved. The UTP semantics of *Circus* was redefined with a deep-embedding of *Circus* in Z, and the theorem prover ProofPower-Z was used to mechanise this semantics an prove the rules. However, the translation strategy that links concrete *Circus* with Java is not formally proved. This is a very complex task, which involves the semantics of both Java and *Circus*. The adequacy of the strategy is evidenced by the successful implementation of the industrial case study and the fairly direct correspondence between the *Circus* constructs and JCSP.

The main motivation behind our work is to contribute with the development of tools for *Circus*. In software engineering, tools are of fundamental importance to make the development process more reliable and efficient. Computer-aided software engineering (CASE) tools offers support to different phases of the software development: from system analysis and design, through tools for project management, compilers, test, and so on. In the field of formal methods, specifically, the use of tools is of particular importance in performing tasks that are impossible for humans due to the amount of data to analyse (for example, model checking) or to automatise repetitive tasks with manipulation of extensive formulas (for example, refinement calculus and theorem proving).

Because of that, we can say that tool support is a key factor in the success of a formal language. A formal language can be expressive and provide useful techniques to efficiently solve a determined problem, but if it does not provide tool support, it is doubted that the language will encounter many adepts among the scientific and industrial community. The success of the CSP language, for example, is greatly due to FDR - a model checker for CSP, which has achieved widespread use in academy and industry.

In the original work on the translation strategy from *Circus* to Java, the industrial case study was carried out by hand; the classes were manually coded. The application of these rules, although simple, is time consuming, because each line of *Circus* programs usually yields several lines of Java code. When developing large systems, we easily reach a very large program. Besides that, the process is error-prone; it is easy to get the variable names incorrect during copy-paste operations, for instance. Automation of the strategy is an advance towards the solution of both these problems.

1.2 Objectives

In this work, we implement the translation strategy from *Circus* to Java; the resulting tool is called *JCircus*. It receives as input a concrete *Circus* specification written in LaTeX markup, automatically applies a series of translation rules, and generates a Java project, which implements the specification.

JCircus should be user-friendly, with simple and intuitive operation that does not require much expertise from the user. The tool should also be complete, in the sense that it implements the whole translation strategy, and sound, so that we can rely on the results it produces. We also aim at a good design that facilitates code maintenance and future extensions of the tool that will be needed due to the inevitable evolution of *Circus*. All these aspects were concerns when developing this work.

To achieve these results we used a structured approach for its development, with the following phases: requirement analysis, design, implementation and testing. The work is documented using UML and *JCircus* is implemented in Java.

As already said, the programs that *JCircus* generates use JCSP; this library, however, does not completely implement CSP. One feature that is not provided by JCSP is multi-way synchronisation. In JCSP, all communications are point-to-point. In order to fill this gap, the translation strategy proposes a protocol to implement multi-synchronisation by means of a controller process that manages the requests for synchronisation. We are also concerned with the correctness of implementation; in this work, we also verify this protocol: we propose a *Circus* model for it and use the *Circus* refinement calculus to demonstrate that it really implements a multi-way synchronisation.

The automation of the translation strategy is feasible because it defines rules that are specific to each kind of *Circus* construct. Therefore, each step in the transformation is non-ambiguous. The translation is basically an operation of transformation between a representation of a program, in *Circus*, to another representation, in Java. It is different from a refinement process, for example, where each step gives a range of possibilities for transformation, and user expertise is required for reasoning.

We expect that *JCircus* can be useful for the industrial and academic community that uses *Circus* in formal development. It provides rapid prototyping for validation of a *Circus* specification, and automatic implementation once the specification has been validated.

The long-term goal of the *Circus* project is to gather all the tools currently under development into an integrated environment for *Circus*, which will provide support for all the development phases. Figure 1.1 describes how these parts can be integrated in the development process. The contribution of the work described in this thesis is the implementation of the translator to Java.

1.3 Overview

In the next chapter we introduce the background necessary to understand the work described in this thesis: the *Circus* language and the JCSP library. In Chapter 3 we present the translation strategy implemented by *JCircus* with the modifications to the original strategy that we considered necessary for the automation.

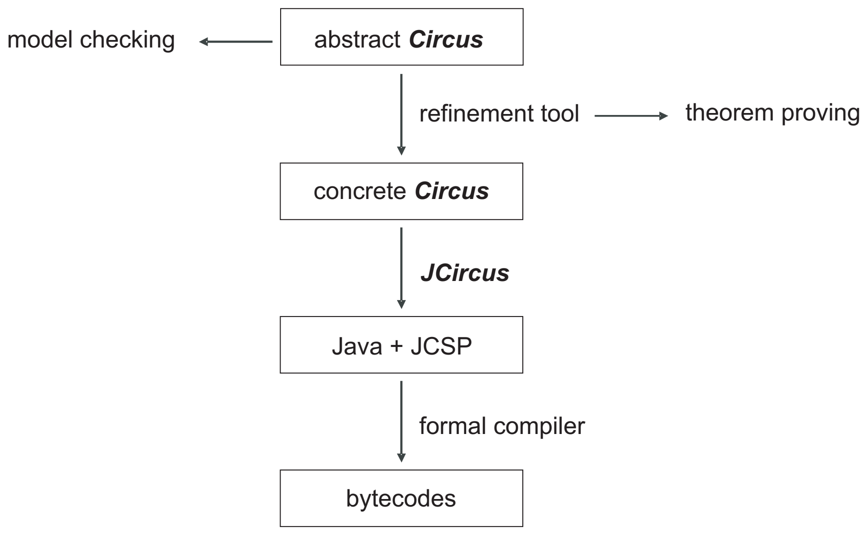


Figure 1.1: Development phases

Chapter 4 describes the tool. Some limitations of JCSP prevent that the strategy can be applied to any format of input specifications. The programs that are subject to translation must comply to a number of requirements, some of them can be achieved by the application of simple refinement rules. In this chapter, we start with an account of such requirements. Then, we describe the architecture of *JCircus* and discuss some design and implementation issues, including the CZT framework which was used in our implementation. We also consider an example of how the tool can be used, and finally, we present the test strategy that was adopted.

During the implementation of the strategy, we discovered some errors in the definitions of some translation rules. These problems are reported in Chapter 5. Alternative implementations to some rules are also presented in this chapter.

Chapter 6 covers the verification of the multi-synchronisation protocol. We propose a *Circus* model for a specific type of multi-synchronisation, and prove that this model and the model for the implementation are related by refinement. Finally, in Chapter 7 we draw conclusions and consider some related work and future work.

The appendices contain complementary material. Appendix A contains the complete set of rules of the translation strategy, and Appendix B presents the *Circus* refinement laws used in the verification of the multi-synchronisation protocol.

Chapter 2

Background

This chapter gives a brief introduction to the *Circus* language [43] and the JCSP library [40]. In section 2.1, we present the *Circus* syntax and explain some of its main constructs. In section 2.2, we present part of the JCSP library that is relevant to the translation strategy.

2.1 *Circus*

Circus is a combination of the well-established notations Z [33] and CSP [21]; it incorporates the schema calculus of Z and the process operators of CSP. It also provides a refinement calculus, extended from the Z Refinement Calculus, which allows a executable program to be derived from an abstract specification in a stepwise fashion.

Like a Z specification, a *Circus* program is formed by a sequence of paragraphs. These paragraphs can be either a Z paragraph, a channel definition, a channel set definition, or a process declaration. Figures 2.1 and 2.2 presents the current *Circus* syntax. The original syntax was firstly published in [43].

We will use a small example of a program that calculates the greatest common divisor (GCD) between two natural numbers to explain some of the main constructs of *Circus*.

Channels are the interface between the system and the external environment. They can be declared as in CSP: the keyword **channel**, the name of the channel, and the type of the values it communicates. In our example, the channels *in* and *out* communicate natural numbers; the former receives the numbers, in sequence, and the latter outputs their GCD.

$$\textbf{channel}\ in, out : \mathbb{N}$$

The type expression can be any Z expression describing sets; the type system of *Circus* follows the type system of Z, with its notion of maximal type [44].

As in CSP, we can also declare untyped channels by omitting the type expression; in such cases, the channel is only a synchronisation event and does not communicate values. *Circus* adds the concept of generic channel [43] (similar to that of generic schemas of Z); and we can also use a schema without a predicative part to declare groups of channels. Channels are always declared in global scope, that is, in the level of *Circus* paragraphs.

A process declaration declares its name and gives a process specification. The most basic form of process specification defines the state of the process, a sequence of process paragraphs, and a nameless main action which describes the behaviour of the process; all these are delimited by the keywords **begin** and **end**. A process paragraph can be a Z paragraph, an action definition, or a name set definition.

$$
\begin{array}{lcl}
\mathsf{Program} & ::= & \mathsf{CircusPara}^* \\[1.5em]
\mathsf{CircusPara} & ::= & \mathsf{Para} \mid \mathbf{channel}\ \mathsf{CDecl} \mid \mathbf{chanset}\ \mathsf{N} == \mathsf{CSExpr} \mid \mathsf{ProcDecl} \\[1.5em]
\mathsf{CDecl} & ::= & \mathsf{SimpleCDecl} \mid \mathsf{SimpleCDecl};\ \mathsf{CDecl} \\[1.5em]
\mathsf{SimpleCDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Expr} \mid [\mathsf{N}^+]\ \mathsf{N}^+ : \mathsf{Expr} \mid \mathsf{Schema\text{-}Exp} \\[1.5em]
\mathsf{CSExpr} & ::= & \{\!|\mathsf{N}^*|\!\} \mid \mathsf{N} \mid \mathsf{CSExpr} \cup \mathsf{CSExpr} \mid \mathsf{CSExpr} \cap \mathsf{CSExpr} \mid \mathsf{CSExpr} \setminus \mathsf{CSExpr} \mid \mathsf{Appl} \\[1.5em]
\mathsf{ProcDecl} & ::= & \mathbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{ProcDef} \mid \mathbf{process}\ \mathsf{N}[\mathsf{N}^+] \mathrel{\widehat=} \mathsf{ProcDef} \\[1.5em]
\mathsf{ProcDef} & ::= & \mathsf{ParamProc} \mid \mathsf{Proc} \\[1.5em]
\mathsf{ParamProc} & ::= & \mathsf{Decl} \bullet \mathsf{Proc} \mid \mathsf{Decl} \odot \mathsf{Proc} \mid \mu\,\mathsf{N} \bullet \mathsf{ParamProc} \\[1.5em]
\mathsf{Proc} & ::= & \mathbf{begin}\ \mathsf{PPara}^*\ \mathbf{state}\ \mathsf{Para}\ \mathsf{PPara}^* \bullet \mathsf{Action}\ \mathbf{end} \\
 & | & \mathsf{Comm} \to \mathsf{Proc} \mid \mathsf{Pred}\ \&\ \mathsf{Proc} \mid \mathsf{Proc} \setminus \mathsf{CSExpr} \mid \mathsf{Proc}[\mathsf{N}^+ := \mathsf{N}^+] \\
 & | & \mathsf{Proc}\ ;\ \mathsf{Proc} \mid \mathsf{Proc} \Box \mathsf{Proc} \mid \mathsf{Proc} \sqcap \mathsf{Proc} \mid \mathsf{Proc} \,|[\, \mathsf{CSExpr} \parallel \mathsf{CSExpr} \,]|\, \mathsf{Proc} \\
 & | & \mathsf{Proc} \,|[\, \mathsf{CSExpr} \,]|\, \mathsf{Proc} \mid \mathsf{Proc} \,|||\, \mathsf{Proc} \mid \mathsf{N} \mid \mathsf{N}(\mathsf{Expr}^+) \mid (\mathsf{Decl} \bullet \mathsf{Proc})(\mathsf{Expr}^+) \\
 & | & (\mu\,\mathsf{N} \bullet \mathsf{Decl} \bullet \mathsf{Proc})(\mathsf{Expr}^+) \mid \mathsf{N}\lfloor \mathsf{Expr}^+ \rfloor \mid (\mathsf{Decl} \odot \mathsf{Proc})\lfloor \mathsf{Expr}^+ \rfloor \\
 & | & (\mu\,\mathsf{N} \bullet \mathsf{Decl} \odot \mathsf{Proc})\lfloor \mathsf{Expr}^+ \rfloor \mid \mathsf{N}[\mathsf{Expr}^+] \mid \mathsf{N}[\mathsf{Expr}^+](\mathsf{Expr}^+) \mid \mathsf{N}[\mathsf{Expr}^+]\lfloor \mathsf{Expr}^+ \rfloor \\
 & | & \mu\,\mathsf{N} \bullet \mathsf{Proc} \mid \,;\, \mathsf{Decl} \bullet \mathsf{Proc} \mid \Box\, \mathsf{Decl} \bullet \mathsf{Proc} \mid \sqcap\, \mathsf{Decl} \bullet \mathsf{Proc} \\
 & | & \parallel \mathsf{Decl} \bullet |[\mathsf{CSExpr}]|\, \mathsf{Proc} \mid |[\, \mathsf{CSExpr} \,]|\, \mathsf{Decl} \bullet \mathsf{Proc} \mid \,|||\, \mathsf{Decl} \bullet \mathsf{Proc} \mid (\mathsf{Proc})
\end{array}
$$

Figure 2.1: *Circus* syntax

In our example (Figure 2.3), the process *GCDEuclides* has its internal state described by the schema *GCDState*. It contains two natural numbers, a and b. These values are initialised with the numbers for which we want to calculate the GCD.

The definitions that follow describe actions. *InitState* and *UpdateState* are both Z schemas. The first initialises the state components. It declares two input variables $x?$ and $y?$; the declaration $GCDState'$ introduces the dashed versions of the state components, that is, a' and b', which represent their final values. *Circus* adopts the same conventions of Z for the decoration of input, output, and state variables. The predicative part states that the final values of a and b are equal to the values of $x?$ and $y?$, respectively. The schema *UpdateState* updates the values of the state components in each iteration of the calculus of the GCD.

The action *GCD* is a recursive action that implements the Euclidean algorithm for the calculus of the greatest common divisor between two natural numbers. This algorithm is based on the fact that $gcd(a, b) = gcd(b, a \mathsf{\ mod\ } b)$. The recursive case, when $b \neq 0$, calls the schema *UpdateState*, and then makes the recursive call. The base case is chosen when the GCD is found (the GCD between a and zero is a): it is output through channel *out*. As in CSP, the basic action *Skip* does not communicate any value nor changes any state; it just terminates immediately.

The main action describes the behaviour of the process. It is defined as a sequential composition of two actions. The first receives two inputs through channel *in* and initialises the state with these values; the second is a call to action *GCD*, which as previously discussed, calculates and outputs

$$
\begin{array}{lll}
\mathsf{PPara} & ::= & \mathsf{Para} \mid \mathbf{nameset}\ \mathsf{N} == \mathsf{NSExpr} \mid \mathsf{N} \mathrel{\widehat=} \mathsf{ActionDef} \\[1.5ex]
\mathsf{NSExpr} & ::= & \{\mathsf{N}^*\} \mid \mathsf{N} \mid \mathsf{NSExpr} \cup \mathsf{NSExpr} \mid \mathsf{NSExpr} \cap \mathsf{NSExpr} \mid \mathsf{NSExpr} \setminus \mathsf{NSExpr} \mid \mathsf{Appl} \\[1.5ex]
\mathsf{ActionDef} & ::= & \mathsf{ParamAction} \mid \mathsf{Action} \\[1.5ex]
\mathsf{ParamAction} & ::= & \mathsf{Decl} \bullet \mathsf{Action} \mid \mu\,\mathsf{N} \bullet \mathsf{ParamAction} \\[1.5ex]
\mathsf{Action} & ::= & \mathsf{CSPAction} \mid \mathsf{Schema\text{-}Exp} \mid \mathsf{ParCommand} \mid (\mathsf{Action}) \\[1.5ex]
\mathsf{CSPAction} & ::= & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \\
 & \mid & \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred}\ \&\ \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CSExpr} \mid \mathsf{Action}[\mathsf{N}^+ := \mathsf{N}^+] \\
 & \mid & \mathsf{Action}\ ;\ \mathsf{Action} \mid \mathsf{Action} \Box \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
 & \mid & \mathsf{Action} \llbracket \mathsf{NSExpresion} \mid \mathsf{CSExpr} \| \mathsf{CSExpr} \mid \mathsf{NSExpr} \rrbracket \mathsf{Action} \\
 & \mid & \mathsf{Action} \llbracket \mathsf{CSExpr} \| \mathsf{CSExpr} \rrbracket \mathsf{Action} \mid \mathsf{Action} \llbracket \mathsf{CSExpr} \rrbracket \mathsf{Action} \\
 & \mid & \mathsf{Action} \llbracket \mathsf{NSExpresion} \mid \mathsf{CSExpr} \mid \mathsf{NSExpr} \rrbracket \mathsf{Action} \\
 & \mid & \mathsf{Action} \ |\!|\![\mathsf{NSExpr} \mid \mathsf{NSExpr}]\!|\!|\ \mathsf{Action} \mid \mathsf{Action} \mathbin{|||} \mathsf{Action} \\
 & \mid & \mathsf{N} \mid \mathsf{N}(\mathsf{Expr}^+) \mid (\mathsf{ParamAction})(\mathsf{Expr}^+) \mid \mu\,\mathsf{N} \bullet \mathsf{Action} \\
 & \mid & \mathbin{;}\ \mathsf{Decl} \bullet \mathsf{Action} \mid \Box\ \mathsf{Decl} \bullet \mathsf{Action} \mid \sqcap\ \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & \|\ \mathsf{Decl} \bullet \llbracket \mathsf{NSExpr} \mid \mathsf{CSExpr} \rrbracket \mathsf{Action} \mid \|\ \mathsf{Decl} \bullet \llbracket \mathsf{CSExpr} \rrbracket \mathsf{Action} \\
 & \mid & \llbracket \mathsf{CSExpr} \rrbracket \mathsf{Decl} \bullet \llbracket \mathsf{NSExpr} \rrbracket \mathsf{Action} \mid \llbracket \mathsf{CSExpr} \rrbracket \mathsf{Decl} \bullet \mathsf{Action} \\
 & \mid & |||\ \mathsf{Decl} \bullet |\!|\![\mathsf{NSExpr}]\!|\!|\mathsf{Action} \mid |||\ \mathsf{Decl} \bullet \mathsf{Action} \\[1.5ex]
\mathsf{Comm} & ::= & \mathsf{N}\ \mathsf{CParam}^* \mid \mathsf{N}\ [\mathsf{Expr}^+]\ \mathsf{CParam}^+ \\[1.5ex]
\mathsf{CParam} & ::= & ?\mathsf{N} \mid ?\mathsf{N} : \mathsf{Pred} \mid !\mathsf{Expr} \mid .\mathsf{Expr} \\[1.5ex]
\mathsf{ParCommand} & ::= & \mathsf{Command} \mid (\mathsf{ParDecl} \bullet \mathsf{Command}) \\[1.5ex]
\mathsf{ParDecl} & ::= & \mathsf{ParQualifier}\ \mathsf{Decl} \mid \mathsf{ParQualifier}\ \mathsf{Decl};\ \mathsf{ParDecl} \\[1.5ex]
\mathsf{ParQualifier} & ::= & \mathbf{val} \mid \mathbf{res} \mid \mathbf{valres} \\
\mathsf{Command} & ::= & \mathsf{N}^+ : [\mathsf{Pred}, \mathsf{Pred}] \mid \mathsf{N}^+ := \mathsf{Expr}^+ \mid \mathbf{if}\ \mathsf{GuardActions}\ \mathbf{fi} \\
 & \mid & \mathbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \mid (\mathsf{ParDecl} \bullet \mathsf{Command})(\mathsf{Expr}^+) \mid \{\mathsf{Pred}\} \mid [\mathsf{Pred}] \\[1.5ex]
\mathsf{GuardActions} & ::= & \mathsf{Pred} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \rightarrow \mathsf{Action} \Box \mathsf{GuardActions}
\end{array}
$$

Figure 2.2: *Circus* syntax

```
process $GCDEuclides \mathrel{\widehat=}$ begin

    state $GCDState \mathrel{\widehat=} [a, b : \mathbb{N}]$
    $InitState \mathrel{\widehat=} [GCDState';\ x?, y? : \mathbb{N} \mid a' = x? \land b' = y?]$
    $UpdateState \mathrel{\widehat=} [GCDState;\ GCDState' \mid a' = b \land b' = a \mathbin{\mathsf{mod}} b]$
    $GCD \mathrel{\widehat=}$
        $\mu X \bullet$ if $b = 0 \rightarrow\ out!a \rightarrow Skip$
                $[\!]\ b \neq 0 \rightarrow\ UpdateState;\ X$
                fi

    $\bullet\ (in?x \rightarrow in?y \rightarrow InitState);\ GCD$
end
```

Figure 2.3: Euclidean algorithm for GCD in *Circus*

the GCD of the given numbers.

The input variables x and y from the communication on channel in are not the same variables as the ones declared in $InitState$; those are implicitly declared at the moment of the communication, with the same type as the channel's type. Their scope is the action that follows the prefixing. In *Circus*, the value passed to an input variable in a schema is the value of the variable in scope with the same name, without taking decoration into account. Therefore, the values taken by the schema input variables $x?$ and $y?$ are the values taken by the prefixing input variables x and y.

A process definition like that of $GCDEuclides$ is the most basic kind of process definition: it uses Z and CSP constructs to define the state and the behaviour of the process. It is also possible to define processes in terms of others previously defined, using the CSP operators for sequence, external choice, internal choice, parallelism and interleaving, among others.

In Figure 2.4, process $SumOrGCD$ is a parallel composition of the process $GCDEuclides$ previously discussed and $GCDClient$. They communicate via channels in and out, and these channels are hidden, which means that the environment cannot see communications through them.

The process $GCDClient$ behaves recursively. In each iteration, it reads values x and y from a channel $read$, and passes them to the parametrised action $ChooseOper$, which offers a choice between the operation of sum or the greatest common divisor. The external choice operator is as in CSP: it offers the environment a choice between two or more actions. If the GCD operation is chosen, it interacts with process $GCDEuclides$ through channels in and out, and then outputs on $write$ the result obtained. Otherwise, it outputs on $write$ the summation of the two values.

Other *Circus* operators for processes include: interleaving ($|||$), which is a special type of parallelism, in which there is no interaction between the parallel processes; internal choice ($\sqcap$), which describe a non-deterministic choice between two processes; indexing ($\odot$), which renames the channels that a process uses according to a set of variables. As in CSP, *Circus* defines iterated versions for the following operators: sequential composition, external choice, internal choice, parallelism and interleaving. The iterated operators are used with parametrised processes to define several instances of a process. For example, $|||\ i : T \bullet P(i)$ is the process defined by interleaving each of the process $P(v)$ formed by instantiating P with a value v of T. A complete description of *Circus* and its semantics can be found in [29].

```
channel $gcd, sum$
channel $read, write : \mathbb{N}$

process $SumOrGCD \mathrel{\widehat=} (GCDEuclides \mathrel{[\![} \{\!|\ in, out\ |\!\} \mathrel{]\!]} GCDClient) \setminus \{\!|\ in, out\ |\!\}$

process $GCDClient \mathrel{\widehat=}$ begin
    $ReadValue \mathrel{\widehat=} read?x \rightarrow read?y \rightarrow ChooseOper(x, y)$
    $ChooseOper \mathrel{\widehat=} x, y : \mathbb{N} \bullet$
        $gcd \rightarrow in!x \rightarrow in!y \rightarrow out?r \rightarrow write!r \rightarrow Skip$
        $\Box$
        $sum \rightarrow write!(x + y) \rightarrow Skip$
    $\bullet\ \mu X \bullet ReadValue;\ X$
end
```

Figure 2.4: Parallelism between *GCDEuclides* and *GCDClient*

2.2 JCSP

Java Communicating Sequential Processes (JCSP) [40] is a Java class library that provides a base range of CSP primitives and a rich set of extensions. The main motivation behind JCSP is to simplify the programming of concurrent systems. Instead of dealing directly with the Java primitives for concurrency, one can use the JCSP library to directly implement a CSP specification in Java, after it has been verified for desirable properties using the techniques and tools available for CSP.

In JCSP, a process is a class that implements the interface `CSProcess`. In Java, an interface is a class that defines methods, but does not implement them. Rather, the class implementing the interface must implement the methods it defines. The interface `CSProcess` defines only the method `public void run()`, which encodes the behaviour of the process.

JCSP also defines interfaces for channels: `ChannelInput` is the interface for input channels and defines the method `read`; `ChannelOutput` is the interface for output channels and defines the method `write`; `Channel` extends both `ChannelInput` and `ChannelOutput` and is used for channels which are not specified as input or output channels. The implementations for channels are the classes `One2OneChannel`, `One2AnyChannel`, `Any2OneChannel` and `Any2AnyChannel`. The appropriate implementation to be used when creating a channel depends on whether there are one or more possible readers and writers for the channel.

Synchronisation in JCSP is not in exact correspondence with the original concept in CSP. Despite being possible to have more than one process that reads or writes on a channel, only one pair reader/writer can synchronise at each time. Thus, multi-synchronisation, that is, three or more processes synchronising on one event, which is allowed in CSP, is not directly supported by JCSP. The translation strategy implements a protocol for multi-synchronisation, which will be explained in the next chapter.

The class `Alternative` implements the external choice operator. Its constructor takes an array of guards, which are the channels that may be selected. The implementation of the alternation requires that only input channels that have at most one reader can participate on it. The abstract class `AltingChannelInput` defines channels that can be used in alternation: it extends only the interface `ChannelInput`, and the implementations provided within the framework are `One2OneChannel` and `Any2OneChannel`. Figure 2.5, which was adapted from [6], presents an overview of the JCSP classes for channels. The diagram is in UML notation (version 1.3): italic

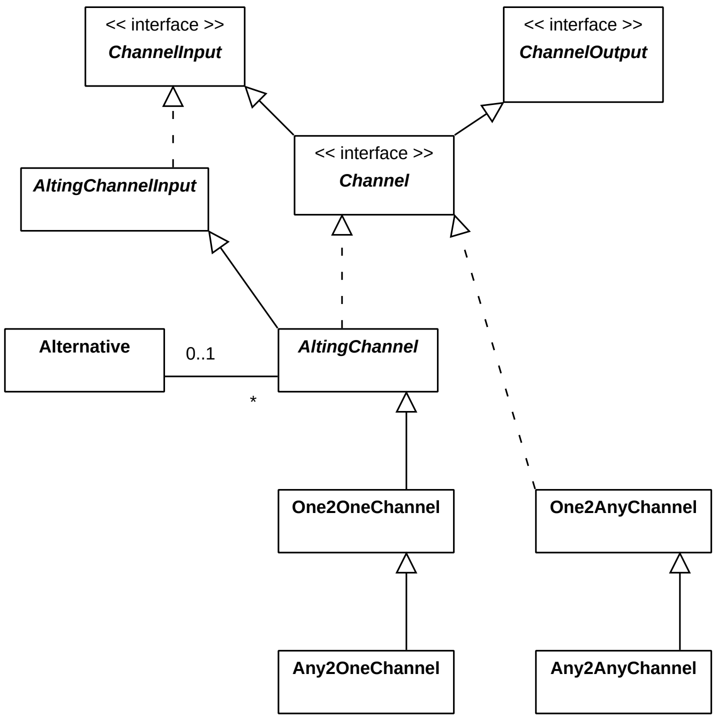


Figure 2.5: JCSP Channel architecture

names represent abstract classes and interfaces; interfaces are represented with and stereotype; an arrow with triangular arrowhead represents the inheritance relationship; a dashed arrow with triangular arrowhead means that the class implements the interface to which it points. For details on the UML notation, see [8].

Here is an example of the use of class `Alternative`:

```
AltingChannelInput l = new Any2OneChannel();
AltingChannelInput r = new Any2OneChannel();
AltingChannelInput[] chs = new AltingChannelInput[] {l, r};
Alternative alt = new Alternative(chs);
chs[alt.select()].read();
```

The alting channels `l` and `r` are instantiated as `Any2OneChannel`s. The array of `AltingChannelInputs` is declared and passed to the constructor of class `Alternative`. The method `select()` waits for one or more channels to become available, makes an arbitrary choice between them, and returns the index of the selected channel. The index is used here to access the channel in the array. Since it is an output channel, the method `read` is called to make the synchronisation happen.

Parallel and interleaved processes are implemented using the class `Parallel`, which implements `CSProcess`. The constructor takes an array of `CSProcess`es, which are the processes that will compose the parallelism. The method `run` executes all processes in parallel and terminates when all processes terminate. There is no difference between the way in which interleaving and parallelism are implemented because, differently from CSP, it is not possible to choose the channels on which the processes synchronise; in JCSP, they synchronise on all channels that they have in common. Therefore, if the intersection of the alphabets is not empty, we have a parallelism; otherwise, we have actually an interleaving.

The CSP constructors *Skip* and *Stop* are implemented by the classes `Skip` and `Stop`, respectively. JCSP includes many other facilities beyond those available in CSP, such as implementations for barrier synchronisation, timers and process managers, among others. We use the `ProcessManager` class in our implementation of multi-synchronisation, which is explained in Section 3.6. There are also plug-and-play components, to emphasize code reuse, and a package providing extensions for all `java.awt` components. These, however, are not used in the translation strategy, and are not going to be presented here. For details, see [40].

2.3 Final considerations

In this chapter, we have briefly introduced the *Circus* language, the JCSP library, with the illustrative example of a program that implements the Euclidean algorithm for the calculus of the GCD. Due to space restrictions we have not presented all features of the *Circus* language and the JCSP library; rather only those essential for the understanding of the basics of the translation strategy, presented in next chapter. More details about *Circus* and JCSP can be found in [43, 40].

Chapter 3

Translation strategy

The first requirement of the translation strategy is that we must have as a starting point a concrete *Circus* specification. In a concrete *Circus* specification, schemas and specification statements must not be used to define actions. Also, guards must be conditions; they must not involve quantifiers. Thus, the specification of the process *GCDEuclides* as stated in the previous chapter must be refined prior to the application of the translation strategy. The refinement in this case is straightforward; basically it consists of the application of a law that refines a schema definition to a (parametrised) action with an assignment in its body (a reference to this refinement law can be found in [29]). Figure 3.1 shows the resulting specification.

The strategy also requires that the specification conform to some requirements; not any *Circus* program is possible to be translated. The requirements are presented and discussed in Chapter 4. Our example is in a suitable format for translation.

The translation strategy considers that some environments are available throughout the translation. In the following definitions, N is the type of names (process, channel, and variable names) and Expr is the type of expressions. The environments $VisChanEnv : \mathrm{seq}\,\mathsf{N}$ and $HidChanEnv : \mathrm{seq}\,\mathsf{N}$ record the names of the visible and hidden channel of a process. The hidden channels are those that are concealed from the environment with the use of the hiding operator, as channels *in* and *out* in process *SumOrGCD* (Figure 2.4). The channels which are not hidden are visible. A channel can be hidden in one process and visible in another process, and that is why it is necessary to keep these environments separate for each process definition. For instance, process $GCDEuclides_1$ uses two channels, *in* and *out* and they are visible; process *GCDClient* uses six channels, *in*, *out*, *read*, *write*, *sum* and *gcd*, and they are all visible; process *SumOrGCD* uses all the above channels, but *in* and *out* are hidden, and the others are visible.

The environment $SyncCommEnv : \mathsf{N} \nrightarrow SC$ maps each channel name used in a process to a value of type SC: S, if it is a synchronisation channel, or C, if it is a communication channel. A communication channel is a channel that defines an input (?) or output (!) field, as channels *in* and *out* of our example. A synchronisation channel is a channel that is either untyped, as channels *sum* and *gcd*, or contains only fields that are not defined as input or output.

The environment $ChanTypeEnv : \mathsf{N} \nrightarrow (\mathrm{seq}\,\mathsf{Expr} \times \mathrm{seq}\,\mathsf{Expr})$ maps each channel in the program to its *Circus* type. The first sequence is the generic names and the second sequence is the type expressions for each channel field. If the channel is not generic, the first sequence is empty. If the channel is untyped, that is, it does not define any field, the second sequence contains only the special type *Sync*.

The environments $LocalVarEnv : \mathrm{seq}(\mathsf{N} \times \mathsf{Expr})$ and $StateCompEnv : \mathrm{seq}(\mathsf{N} \times \mathsf{Expr})$ map a local variable, or a state component, to its *Circus* type. They are used in the translation of parallel and recursive actions. The environment $TypesEnv : \mathrm{seq}\,\mathsf{Expr}$ is a list of all *Circus* types used in

$$
\begin{array}{l}
\textbf{process } \mathit{GCDEuclides}_1 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state } \mathit{GCDState} \mathrel{\widehat=} [a, b : \mathbb{N}] \\
\quad \mathit{InitState} \mathrel{\widehat=} x, y : \mathbb{N} \bullet a, b := x, y \\
\quad \mathit{UpdateState} \mathrel{\widehat=} a, b := b, a \bmod b \\
\quad \mathit{GCD} \mathrel{\widehat=} \mu X \bullet \textbf{if } b = 0 \rightarrow \ \mathit{out}!a \rightarrow \mathit{Skip} \\
\qquad\qquad\qquad\quad [\!] \ b \neq 0 \rightarrow \ \mathit{UpdateState};\ X \\
\qquad\qquad\qquad\quad \textbf{fi} \\
\quad \bullet\ \mathit{in}?x \rightarrow \mathit{in}?y \rightarrow \mathit{InitState}(x, y);\ \mathit{GCD} \\
\textbf{end}
\end{array}
$$

Figure 3.1: Concrete *Circus* program for calculation of the GCD.

the *Circus* program being translated.

The environment $\mathit{ReadWriteEnv} : \mathsf{N} \leftrightarrow (\mathsf{N} \leftrightarrow \mathit{ChanUse})$ records how a channel is used within the sub-processes of a process. It is used to decide which processes read and which processes write on a channel. It maps a channel name to a function which associates a process name to a value of type $\mathit{ChanUse} ::= I \mid O$. For instance, in our GCD example, the environment *ReadWriteEnv* for process *SumOrGCD* records the following:

$$
\begin{array}{l}
\{(\mathit{in} \mapsto \{(\mathit{GCDEuclides}_1 \mapsto I), (\mathit{GCDClient} \mapsto O)\}), \\
(\mathit{out} \mapsto \{(\mathit{GCDEuclides}_1 \mapsto O), (\mathit{GCDClient} \mapsto I)\})\}
\end{array}
$$

Process *SumOrGCD* is a parallelism of processes $\mathit{GCDEuclides}_1$ and *GCDClient*. The channel *in* is an input channel in process $\mathit{GCDEuclides}_1$ and output channel in process *GCDClient*. For process *out*, it is the other way around.

The environment $\mathit{MultiSyncEnv} : \mathsf{N} \leftrightarrow (\mathsf{N} \rightarrowtail \mathbb{N})$ records information for the multi-synchronisation protocol. It records an identification number for each process that takes part in a multi-synchronisation. This environment will be better explained in Section 3.6.

The strategy also defines some auxiliary functions to be used in the definition of the rules. The function *JType* maps a *Circus* type expression into the name of the Java class that represents that type. The function *JExp* translates expressions. As an example of the use of these auxiliary functions, we have that $\mathit{JType}(\mathbb{Z}) =$ `CircusNumber` and $\mathit{JExp}(x > y) =$ `x.greaterThan(y)`. The class `CircusNumber` is an auxiliary class that we have implemented to represent the *Circus* built-in type $\mathbb{A}$ (pronounced "arithmos"), which represents number types. We are considering a simplified type system, which includes only free types and number types.

The output of the translation is Java code composed of several classes allocated in four packages. For each program, a project name *proj* is required. The package `proj` contains the classes with the method `main` to be executed; `proj.axiomaticDefinitions` contains the class with the translation of the axiomatic definitions declared in global scope, that is, in the level of *Circus* paragraphs; package `proj.processes` contains the classes that result from the translation of each process declaration; and package `proj.typing` contains the classes that implement types.

3.1 Process declarations

Each process declaration is translated into a Java class that implements the JCSP interface `jcsp.lang.CSProcess`. The Java class has the same name as the process, and imports the necessary packages: the Java utility package, the basic JCSP package, and some project packages.

Rule A.1 *Normal process declaration*

```
$[\![$ process $P \mathrel{\widehat=} ProcDef\ ProcDecls\,]\!]^{ProcDecl}\ proj =$
    package $proj$.processes;
    import java.util.*;
    import jcsp.lang.*;
    import $proj$.axiomaticDefinitions.*;
    import $proj$.typing.*;
    public class P implements CSProcess { $[\![\, ProcDef\,]\!]^{ProcDef}$ P }
    $[\![\, ProcDecls\,]\!]^{ProcDecls}\ proj$
```

Using Rule A.1, we translate the process $GCDEuclides_1$. The translation is shown below and we omit package and import declarations shown in the rule.

```
public class GCDEuclides_1 implements CSProcess {
    $[\![$ begin state $GCDState \mathrel{\widehat=} ... \bullet in?x \rightarrow in?y \rightarrow \ldots$ end $]\!]^{ProcDef}\ GCDEuclides_1$
}
```

The next step is to translate the process definition. We use the rule below for non-parametrised process definitions.

Rule A.3 *Non-parametrised process definition*

```
$[\![Proc\,]\!]^{ProcDef}\ P\ =$
    $ChannelDecl\ VisChanEnv\ ChanTypeEnv\ SyncCommEnv$
    $ChannelDecl\ HidChanEnv\ ChanTypeEnv\ SyncCommEnv$
    public $P$(($VisibleCArgs\ VisChanEnv\ ChanTypeEnv$
                                 $SyncCommEnv$)){
        $MultiAssign\ (ChannelDecl\ VisChanEnv\ ChanTypeEnv$
                                 $SyncCommEnv)$
                     $(VisibleCArgs\ VisChanEnv\ ChanTypeEnv$
                                 $SyncCommEnv)$
        $HiddenCCreation\ HidChanEnv\ ChanTypeEnv$
                     $SyncCommEnv\ TypesEnv\ MultiSyncEnv\ ReadWriteEnv\ P$
    }
    public void run(){  $ProcessCall\ Proc\ HidChanEnv\ MultiSyncEnv$  }
```

This rule uses the auxiliary function *ChannelDecl* to declare as private attributes the visible and hidden channels that the process uses. It takes the environments *ChanTypeEnv* and *SyncCommEnv* to decide if they will be declared as simple channels or as an array of channels. An array of channels is used for some special types of channels. This will be explained in Section 3.7.

Visible channels are taken by the constructor as parameters; the function *VisibleCArgs* declares the parameters and *MultiAssign* initialises them. Hidden channels are instantiated within the constructor (function *HiddenCCreation*). The definition of these auxiliary functions can be found in Appendix A (Rule A.3). Table 3.1 shows how each channel is used in the processes defined in our GCD example.

In the translation strategy, we do not use the channel implementations provided by JCSP directly. Instead, we define a class `GeneralChannel`, which encapsulates an `Any2OneChannel`,

Process	**Channels used**	**Visible channels**	**Hidden channels**
$GCDEuclides_1$	in, out	in, out	None
$GCDClient$	All	All	None
$SumOrGCD$	All	$read$, $write$, gcd, sum	in, out

Table 3.1: Use of channels

an array of `Any2OneChannel` that is used for the multi-synchronisation protocol, and some other information. The class `GeneralChannel`, which also defines methods `read` and `write`, will be explained in detail in the Section 3.6. The creation of this class was one of our modifications to the original translation strategy.

In our example, the process $GCDEuclides_1$ uses two channels, in and out, and they are visible. We proceed with the translation, which is shown below.

```
public class GCDEuclides_1 implements CSProcess {

    private GeneralChannel in;
    private GeneralChannel out;

    public GCDEuclides_1(GeneralChannel in, GeneralChannel out) {
        this.in = in;
        this.out = out;
    }
    public void run () { $[\![\mathbf{begin\ state}\ GCDState \mathrel{\widehat=} \ldots \bullet \ldots \mathbf{end}]\!]^{Proc}$ }
}
```

The method `run` implements the process body translated by $[\![\_]\!]^{Proc}$. This function translates a basic process into a call to the method `run` of an instance of an anonymous inner class that implements `CSProcess`. An anonymous inner class in Java is a nameless class defined inside another class. It is defined using the constructor of a superclass or an interface. The use of this Java feature is justified to allow compositional translation so that we do not have to name a process to translate it. The rule for basic processes is as follows.

Rule A.6 *Basic process*

```
$[\![\mathbf{begin}\ PPars_1\ \mathbf{state}\ PSt\ PPars_2 \bullet Main]\!]^{Proc} =$
    (new CSProcess(){
        $(StateDecl\ PSt)$    $[\![PPars_1\ PPars_2]\!]^{PPars}$
        public void run() {  $[\![Main]\!]^{Action}$  }
    }).run();
```

The inner class declares the state components as private attributes. The function $[\![\_]\!]^{PPars}$ translates the process paragraphs, which can be axiomatic definitions, parametrised or non-parametrised action definitions. Because the process paragraphs can only be referenced within the basic process where they are defined, they are all translated into private methods.

The body of the method `run` is the translation of the main action. In our example, after the application of Rule A.6, we get the following class:

```
public class GCDEuclides_1 implements CSProcess {

    private GeneralChannel in;
    private GeneralChannel out;

    public GCDEuclides_1(GeneralChannel in, GeneralChannel out) {
        this.in = in;
        this.out = out;
    }

    public void run () {
        (new CSProcess() {
            private CircusNumber a;
            private CircusNumber b;
            $[\![\, InitState \mathrel{\widehat=} x, y : \mathbb{N} \bullet a := x;\ b := y\ \ UpdateState \mathrel{\widehat=} ...\, ]\!]^{PPars}$
            public void run(){ $[\![\ \ in?x \rightarrow in?y \rightarrow InitState(x, y);\ \ GCD\ \ ]\!]^{Action}$ }
        }).run();
    }
}
```

The translation of axiomatic definitions is explained in the next section. Parametrised and non-parametrised actions are translated with Rule A.26 and Rule A.25, respectively.

Rule A.26 *Parametrised action definition*

```
$[\![N \mathrel{\widehat=} (Decl \bullet Action)\ PParags]\!]^{PParags} =$
    private void N($ParamsArgs\ Decl$){  $[\![\, Action\, ]\!]^{Action}$  }
    $[\![PParags]\!]^{PParags}$
```

Rule A.25 *Non-parametrised action definition*

```
$[\![N \mathrel{\widehat=} Action\ PParags]\!]^{PParags} =$
    private void  N(){  $[\![\, Action\, ]\!]^{Action}$  }
    $[\![PParags]\!]^{PParags}$
```

Parameters are declared as arguments of the method (function *ParamsArgs*), and the body of the method is the translation of the action, which may be a CSP action or a command.

3.2 Actions

There are several rules for actions, and we will present here the ones we need to continue the translation of our example. Single assignments are directly translated into Java assignments. The function *JExp*, as explained before, translates expressions.

Rule A.51 *Single assignment*

$$[\![x := e]\!]^{Action} = \texttt{x = } (\mathit{JExp}\ e)\texttt{;}$$

Multiple assignments need auxiliary variables in the case where one of the assigned variables also appear in one of the right-hand side expressions. First, the auxiliary variables are declared with the same type as the original value, and they are assigned the translation of the expression. Then they are assigned to the original variables. The function FV gives the set of free variables of the expression that it takes as argument.

Rule A.52 *Multiple assignment*

```
$[\![x_1, \ldots, x_n := e_1, \ldots, e_n]\!]^{Action} =$
    **if** $(\{x_1, \ldots, x_n\} \cap (FV(e_1) \cup \ldots \cup FV(e_n)) = \varnothing)$ **then**
        x_1=$(JExp\ e_1)$; ...; x_n=$(JExp\ e_n)$;
    **else**
        $(JType\ (CType\ x_1))$ aux_x_1 = $(JExp\ e_1)$;
        ...;
        $(JType\ (CType\ x_n))$ aux_x_n = $(JExp\ e_n)$;
        x_1=aux_x_1;
        ...;
        x_n=aux_x_n;
```

Using Rule A.26 (Parametrised action definition), Rule A.25 (Non-parametrised action definition), Rule A.51 (Single assignment) and Rule A.52 (Multiple assignment), the translation of actions *InitState* and *UpdateState* yields:

```
private void InitState(CircusNumber x, CircusNumber y){
    a = x;
    b = y;
}

private void UpdateState(){
    CircusNumber aux_a = b;
    CircusNumber aux_b = a.mod(b);
    a = aux_a;
    b = aux_b;
}
```

A recursive action is translated using an inner class that implements the body of the recursion, and a call to the method `run` of the inner class. The inner class declares copies of the local variables (*DeclLocalVars*), and their values are given to the constructor as arguments (*LocalVarsArg*). The method `run` of this inner class executes the body of the recursion and the function *RenameVars* substitutes the names of local variables by the names of their copies. At the point where the recursive call occurs, it instantiates a new object of this class and executes it. The function *RunRecursion*, at the end, instantiates a recursive process, invokes its method `run`, and collects the values of the auxiliary variables.

Rule A.43 *Recursive action*

```
$[\![\mu\, X \bullet Action(X)]\!]^{Action}$ =
    class I_$index$ implements CSProcess {
        $DeclLocalVars\ LocalVarEnv\ index\ L$
        public I_$index$($LocalVarsArg\ LocalVarEnv$) {
            $InitLocalVars\ LocalVarEnv\ index\ L$
         }
        public void run() {
            $RenameVars$
                $[\![Action(\texttt{(}RunRecursion\ index\ \langle\rangle\texttt{)})]\!]^{Action}$
                $(SetFirst\ LocalVarEnv)\ index\ L$
        }
    };
    $RunRecursion\ index\ \langle\rangle$
```

An index is used in the name of the inner class in order to avoid name clashes in the case, for example, where there are nested recursions or recursions in sequence. For example, the action $A \mathrel{\widehat=} (\mu\ X \bullet B(X));\ (\mu\ X \bullet C(X))$ defines two recursions in sequence; an inner class `I_0` is declared for the first recursion, and another inner class `I_1` is declared for the second one.

Next, we exemplify the rule for recursive process with the translation for action *GCD* in our example. Here, there are no local variables to copy in the inner class.

```
private void GCD() {

    class I_0 implements CSProcess {

        public I_0() {}
        public void run() {
            $RenameVars\ \ [\![\,(\mathbf{if}...\mathbf{fi})(RunRecursion\ index\ \langle\rangle)]\!]^{Action}$
                    $(SetFirst\ LocalVarEnv)\ index\ L$
        }
    }
    I_0 i_0_0 = new I_0();
    i_0_0.run();
}
```

Circus if-commands are translated into Java if-then-else statements. A *Circus* if-command executes any of the guards that is true. In our implementation, this non-determinism is removed: the first guard that is true is chosen. If none of the guards is true, the action has an undefined behaviour, which we implement as an infinite loop.

Rule A.53 *If-command*

$[\![\mathbf{if}\ g_1 \rightarrow A_1 \Box \ldots \Box\ g_n \rightarrow A_n\ \mathbf{fi}]\!]^{Action} =$

```
if((JExp g1)){
    [[A1]]Action
} else if (...) {
    ...
} else if((JExp gn)){
    [[An]]Action
} else { while(true){} }
```

The rules for communications use the methods `read` and `write` of class `GeneralChannel`. As the scope of an input variable in *Circus* is the action that follows the communication, we translate an input communication by introducing a variable block to declare the input variable. A cast converts the object transmitted in the channel into the appropriate type.

Rule A.32 *Prefixing action – input*

$[\![c?x \rightarrow Action]\!]^{Action} =$ `{` $commType$ `x = (`$commType$`)c.read();` $[\![Action]\!]^{Action}$ `}`

where $commType = JType(last\ (snd\ (ChanTypeEnv\ c)))$

Rule A.33 *Prefixing action – output*

$[\![c!e \rightarrow Action]\!]^{Action} =$ `c.write(`$JExp\ e$`);` $[\![Action]\!]^{Action}$

Action invocations are translated into method calls. Sequential compositions are translated into Java sequential compositions. The basic action *Skip* uses the JCSP class with the same name.

Rule A.44 *Action call*

$[\![N]\!]^{Action} =$ `N();`

Rule A.46 *Parametrised call*

$[\![N(e_1, \ldots, e_n)]\!]^{Action} =$ `N((`$JExp\ e_1$`),...,(`$JExp\ e_n$`));`

Rule A.38 *Sequential composition*

$[\![Action_1;\ Action_2]\!]^{Action} = [\![Action_1]\!]^{Action}\ ;\ [\![Action_2]\!]^{Action}$

Rule A.28 *Skip*

$[\![Skip]\!]^{Action} =$ `(new Skip()).run();`

Using Rule A.53 (If-command), Rule A.33 (Prefixing action – output), Rule A.44 (Action call), Rule A.38 (Sequential composition) and Rule A.28 (Skip), we can complete the translation of action *GCD*:

```
private void GCD() {
    class I_0 implements CSProcess {
        public I_0() {}
        public void run() {
            if ((b.getValue() == (new CircusNumber(0)).getValue())){
                out.write(a);
                (new Skip()).run();
            } else if (b.getValue() != (new CircusNumber(0)).getValue()){
                UpdateState();
                I_0 i_0_0 = new I_0();
                i_0_0.run();
            } else{
                while(true){}
            };
        }
    }
    I_0 i_0_0 = new I_0();
    i_0_0.run();
}
```

In the body of the method `run` is the translation of the if-command. It first checks the value of variable `b`; if it is zero, it outputs the value of the GCD and terminates; otherwise, it calls the method `UpdateState` and makes the recursive call (a new instantiation and execution of the inner class).

Figure 3.2 shows the complete code that results from the translation of process $GCDEuclides_1$. Lines 39-43 show the translation of the main action of process $GCDEuclides_1$. This translation applies Rule A.32 (Prefixing action input) twice, and then, Rule A.38 (Sequential composition), Rule A.46 (Parametrised action call) and Rule A.44 (Action call), in this order.

The strategy defines rules for *Stop*, *Chaos*, external and internal choice of actions, action parallelism, guarded actions and more. Some of these will be discussed in Chapter 5.

```
public class GCDEuclides_1 implements CSProcess {
 private GeneralChannel in, out;
 public GCDEuclides_1(GeneralChannel in, GeneralChannel out) {
  this.in = in;    this.out = out;
 }
 public void run () {
  (new CSProcess() {
   private CircusNumber a, b;
   private void InitState(CircusNumber x, CircusNumber y) {
    a = x;    b = y;
   }
   private void UpdateState() {
    CircusNumber aux_a = b;
    CircusNumber aux_b = a.mod(b);
    a = aux_a;    b = aux_b;
   }
   private void GCD() {
    class I_0 implements CSProcess {
     public I_0() {}
     public void run() {
      if ((b.getValue() ==
          (new CircusNumber(0)).getValue())) {
       out.write(a);
       (new Skip()).run();
      } else if (b.getValue() !=
          (new CircusNumber(0)).getValue()) {
       UpdateState();
       I_0 i_0_0 = new I_0();
       i_0_0.run();
      } else {
       while(true){}
      };
     }
    }
    I_0 i_0_0 = new I_0();
    i_0_0.run();
   }
   public void run() {
    {  CircusNumber x = (CircusNumber) in.read();
       {  CircusNumber y = (CircusNumber) in.read();
          InitState(x, y);
          GCD();
       } }
   }
  }).run();
 }
}
```

Figure 3.2: Translation of process *GCDEuclides*$_1$

3.3 Compound processes

The rules for compound processes define the function $[\![\_]\!]^{Proc}$, which returns a Java command that invokes the execution of the method `run` of objects of the class `CSProcess`. We have already seen one example of a rule that defines the result of this function for a basic process (Rule A.6). The rules for compound processes are mostly similar to the corresponding ones for compound actions. Figure 3.3 shows the result of the translation of the process *SumOrGCD* (without package and import declarations). This translation applies Rule A.1 (Normal process declaration), Rule A.3 (Non-parametrised process definition), Rule A.15 (Hiding) and Rule A.18 (Process parallelism), in this order.

In *SumOrGCD*, channels *in* and *out* are hidden, so they are initialised in the constructor. The object `chInfo_in` is an object of class `ChannelInfo` that is taken as argument by the constructor of `GeneralChannel`. Its motivation will be explained in subsequent sections and chapters. For the moment, let us only say that the constructor of `GeneralChannel` takes a new `Any2OneChannel`, the `ChannelInfo` and the name of the process where it is being initialised.

The parallelism is translated with the class `Parallel` of JCSP, which takes an array of `CSProcesses`. In our example, they are the translation of the calls to $GCDEuclides_1$ and *GCDClient*.

The rule for translation of a process call is similar to the one for an action call. However, the rule for process call needs to pass as parameters the channels that the process requires. In our implementation, we construct new `GeneralChannels` from the current one, changing one attribute which is the name of the process.

All the rules for compound processes are in Appendix A (Rules A.16 to A.22). Some of them will be further discussed in Chapter 5.

3.4 Free types and axiomatic definitions

Z allows the user to define types used within a program with the constructs for given type and free types. Given types define the name of a type between brackets; for example, [*NAME*] defines a basic type called *NAME*, but does not determine any element of this type. Free types, on the other hand, declare, at the same time, the type and its elements. For instance, the definition $PrimaryColours ::= blue \mid red \mid yellow$, declares a type called *PrimaryColours* and its three elements; a variable of type *PrimaryColours* can only hold one of those three values that the definition introduces.

Given types are not considered in our translation strategy, because they do not define components. Therefore, a specification to be translated must not define given types. Free types yield Java classes that represent types. They generate a part of the package `proj.typing`. All class types extends from an abstract class `Type`. This class contains: an integer attribute `value`, which stores the number that represents an element within the type; and public integer constants for each type in the system. The constants for types, and the attributes for values are used in the translation of generic channels and channels with multiple fields. These are translated as an array of `GeneralChannels`, and the integer values are used as indexes of the array. The translation is explained in Section 3.7.

Axiomatic definitions are Z paragraphs that define constants. They contain a declarative part that introduces one or more constants, and a predicative part that defines an invariant, a constraint on the declared constants. For instance, the following axiomatic definition

$$\begin{array}{|l}
a, b : \mathbb{N} \\
\hline
a > b
\end{array}$$

```
public class SumOrGCD implements CSProcess {
  private GeneralChannel  gcd, read, sum, write, in, out;
  public SumOrGCD(GeneralChannel gcd, GeneralChannel read,
                    GeneralChannel sum, GeneralChannel write) {
    // Visible channels
    this.gcd = gcd;
    this.read = read;
    this.sum = sum;
    this.write = write;

    // Hidden channels
    ChannelInfo chInfo_in = new ChannelInfo();
    chInfo_in.put("GCDClient", new Integer(0));
    chInfo_in.put("GCDEuclides_1", new Integer(1));
    this.in = new GeneralChannel(new Any2OneChannel(), chInfo_in, "SumOrGCD");

    ChannelInfo chInfo_out = new ChannelInfo();
    chInfo_out.put("GCDClient", new Integer(0));
    chInfo_out.put("GCDEuclides_1", new Integer(1));
    this.out = new GeneralChannel(new Any2OneChannel(), chInfo_out, "SumOrGCD");
  }
  public void run(){
    new Parallel(new CSProcess[] {
          new GCDClient(new GeneralChannel(gcd, "GCDClient"),
                    new GeneralChannel(in, "GCDClient"),
                    new GeneralChannel(out, "GCDClient"),
                    new GeneralChannel(read, "GCDClient"),
                    new GeneralChannel(sum, "GCDClient"),
                    new GeneralChannel(write, "GCDClient")
          ),
          new GCD(new GeneralChannel(in, "GCDEuclides_1"),
                 new GeneralChannel(out, "GCDEuclides_1")
          )
    }).run();
  }
}
```

Figure 3.3: Translation of *SumOrGCD*

declares two constants a and b, of natural type, and states that a is always greater than b. This is the only constraint, and their values are not defined.

In *Circus*, axiomatic definitions can occur in global scope or as a process paragraph. In any case, we restrict the formats of axiomatic definitions dealt by the translation strategy to those that define only one constant, and define its value in an equality in the predicate part. The axiomatic definitions in global scope are translated as `public static` methods in class `AxiomaticDefinitions` in package `proj.axiomaticDefinitions`, so any part of the program has access to them. The modifier `static` for attributes in Java states that the attribute is a class attribute, instead of an instance attribute. This means that it can be accessed directly from the name of the class, and it is not necessary to instantiate a class to have access to it. Rule A.57 defines the translation of global axiomatic definitions.

Rule A.57 *Global axiomatic definition*

$$\begin{array}{l}[\![v : T \mid v = e_1 \; AxDefs]\!]^{AxDefs} = \\ \quad \texttt{public static } (JType\ T)\ v\texttt{() \{ return } (JExp\ e_1)\texttt{; \}} \\ \quad [\![AxDefs]\!]^{AxDefs}\end{array}$$

Axiomatic definitions in the level of process paragraphs are not visible outside the basic process; therefore, they are translated as a private method in the process class. The translation rule is similar to the previous one. The only difference is that the methods are declared as `private` instead of `public static`.

3.5 *Circus* programs

The function $[\![\_]\!]^{Program}$ summarises the translation strategy. It receives a whole *Circus* program and the project name, and creates: the classes for types; the class `AxiomaticDefinitions`; and the classes for each of the process declarations.

Rule A.58 *Circus program*

$$\begin{array}{l}[\![FreeTypes \; AxDefs \; ChanDecls \; ProcDecls]\!]^{Program} \; proj = \\ \quad DeclareTypeClass \; (FreeTypes \; AxDefs \; ChanDecls \; ProcDecls) \; proj \\ \quad [\![FreeTypes]\!]^{FreeTypes} \; proj \\ \quad DeclareAxDefClass \; proj \; AxDefs \\ \quad [\![ProcDecls]\!]^{ProcDecls} \; proj\end{array}$$

The function *DeclareTypeClass* creates the class `Type`, a superclass for all the types in the specification. The function $[\![\_]\!]^{FreeTypes}$ creates the classes for the free types. The function *DeclareAxDefClass* creates the class for the global axiomatic definitions. And finally, $[\![\_]\!]^{ProcDecls}$, as previously discussed, translates each process declaration.

To compile the project, some additional components are required (for example, the class `RandomGenerator`, used in the translation of the internal choice operation). These components are added as a utility library to the generated project. This utility library is called *JCircusUtil*.

In the next sections we explain the implementation of the multi-synchronisation protocol and the translation of specific types of channels.

3.6 Multi-synchronisation

Multi-synchronisation is implemented using a centralised solution. Suppose we have a set of processes $P_1, \ldots, P_n$, which synchronise on a channel c. The multi-synchronisation is managed by another process, the controller, which handles the requests for multi-synchronisation made by each process P_i. The controller communicates with the processes via simple synchronisations.

There are two main components in the multi-synchronisation protocol. The controller is an instance of the class `MultiSyncControl` which implements `CSProcess`. There must be one controller for each channel involved in multi-synchronisation. This process runs in parallel with the other processes in the network. Each time a process P_i wants to engage in a multi-synchronisation, it must instantiate and execute an object of the class `MultiSyncClient`.

In Figure 3.4 (taken from [29]), we illustrate an architecture using these components where two channels (c_1 and c_2) and four processes (P_0, P_1, P_2 and P_3) are involved in multi-synchronisation. We have one instance of `MultiSyncControl` for each channel, and each process instantiates its own `MultiSyncClient`. The controllers use arrays of channels *from* (*from_c*1 or *from_c*2) to communicate with each of their clients. They are represented by double lines at either side of the picture. The clients share the channels *to* (*to_c*1 or *to_c*2) to communicate with their controller. The channels *to* are not multi-synchronised, since the controllers communicate with one client at a time.

The three top-most clients synchronise on c_1, controlled by the right controller, and the three bottom clients synchronise on c_2, controlled by the left controller. Each of the clients has a different identification regarding each of the controllers. For instance, P_1 is identified as client 0 on the left, and as client 1 on the right. If a value is being passed on the channel, there must be one writer only, and this has to be the client with identification number zero.

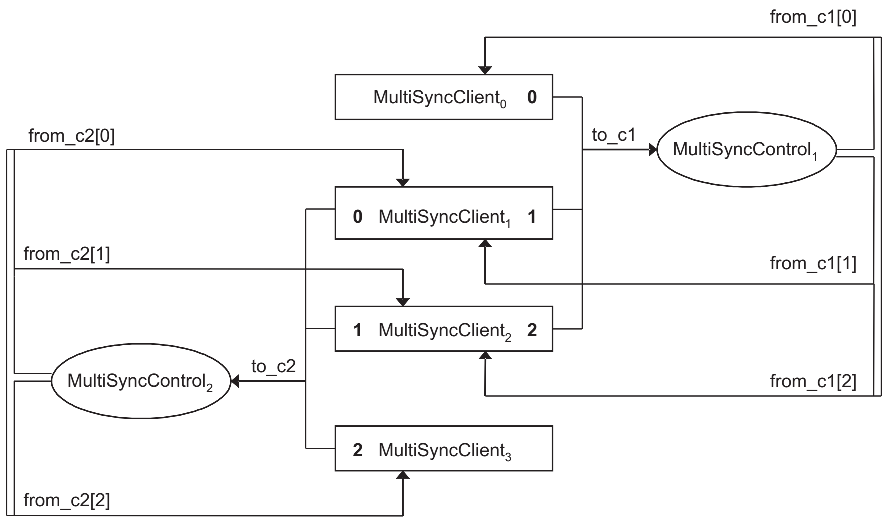


Figure 3.4: Multi-synchronisation components

The environment *MultiSyncEnv* : $\mathsf{N} \nrightarrow (\mathsf{N} \rightarrowtail \mathbb{N})$, mentioned before, returns a function for every name of channel involved in a multi-synchronisation. This function maps the name of every process involved in the multi-synchronisation into its identification number regarding that multi-synchronisation. In our example, the function returned for the channel c_1 is $\{P_0 \mapsto 0, P_1 \mapsto 1, P_2 \mapsto 2\}$; the function for channel c_2 is $\{P_1 \mapsto 0, P_2 \mapsto 1, P_3 \mapsto 2\}$.

Every channel is instantiated within the process that hides it; otherwise, it is received and initialised in the constructor. The controller for a channel c is initialised in the same class where c is initialised. The constructor of class `MultiSyncControl` takes the array of channels *from* and the channel *to*. The number of clients, which determines the size of the array *from*, is retrieved from the environment; it is the cardinality of the function for the channel: $\#(\mathit{MultiSyncEnv}\ c)$. In our example, it is 3 for both c_1 and c_2.

```
Any2OneChannel[] from_c_1 = Any2OneChannel.create(3);
Any2OneChannel to_c_1 = new Any2OneChannel();
MultiSyncControl c_1 = new MultiSyncControl(from_c_1, to_c_1);
```

The instantiation of the clients requires more information. An array of objects `sync` is initialised with: the channel *from* that the controller uses to communicate with this client; the channel *to* that the client uses to communicate with the controller; and the identification of the process in this multi-synchronisation. In our example, in the translation of process P_0, we create the following synchronisation object for channel c_1.

```
Object[] sync = new Object[]{from_c[0], to_c, 0};
```

The multi-synchronisation object is added to a `Vector`

```
Vector seqOfSync = new Vector();
seqOfSync.addElement(sync);
```

and this vector is passed as argument to the constructor of the client.

```
MultiSyncClient client =
    new MultiSyncClient(seqOfSync, seqOfNotSync, v);
client.run();
```

The object `seqOfNotSync` is a `Vector` of `Any2OneChannel` and it is used when the multi-synchronisation takes part in an external choice. The implementation supports choice between multi-synchronisations and simple synchronisations. In this case, there will be one client for the whole external choice: the synchronisation objects for the multi-synchronised channels are added to vector `seqOfSync` and the channels that are not multi-synchronised are added to vector `seqOfNotSync`. When a multi-synchronisation does not occur in an external choice, the constructor receives an empty vector in place of `seqOfSync`.

The last argument `v` is the value communicated through the channel. In the case where the channel does not take part in an external choice, 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, then this argument is null. In the case of an external choice, this element is always null; as explained before, output communications cannot take part in external choices. It is possible to retrieve the value passed in the multi-synchronisation with the method `getValueTrans`.

The initialisation and execution of the client should occur in the moment where the process wants to engage in the multi-synchronisation, that is, when a process wants to read or write on a channel. In our implementation, the instantiation of the client is encapsulated in the methods `read` and `write` of `GeneralChannel`. In the case where the multi-synchronisation happens as part of an external choice, then there is an explicit instantiation of the client.

The controller is implemented using an infinite loop; after completing a cycle for a multi-synchronisation, it goes back to the beginning, and waits for the next request. As a parallelism only terminates when all the parallel process terminates, if we had a simple parallelism between the process and the controller, then we would never reach termination. To solve this problem, the implementation makes use of two other components: the `ProcessManagerMultiSync` and the `ControllersManager`. These two processes run in parallel, and a channel `endManager` is used by the `ProcessManagerMultiSync` to signal to the `ControllersManager` that the process body has terminated. Then the `ControllersManager` stops the execution of every controller it is responsible for. The `ControllersManager` makes use of the class `ProcessManager` from the JCSP library to achieve this implementation.

This behaviour is captured by the function *ProcessCall*. It takes a process definition and checks if it takes part in a multi-synchronisation (that is, $(\operatorname{ran} \mathit{HidChanEnv} \cap \operatorname{dom} \mathit{MultiSyncEnv}) \neq \varnothing$. If it does not, a simple translation of the process is enough; otherwise, it is necessary to call the rules for translation of the components discussed above.

$$
\begin{array}{l}
\mathit{ProcessCall} : \mathsf{ProcDef} \pfun \operatorname{seq} \mathsf{N} \pfun (\mathsf{N} \pfun (\mathsf{N} \pfun \mathbb{N})) \pfun \mathit{JCode} \\
\mathit{ProcessCall}\ \mathit{Proc}\ \iota\ \omega = \\
\quad \mathbf{if}\ (\operatorname{ran} \iota \cap \operatorname{dom} \omega) \neq \varnothing\ \mathbf{then} \\
\quad\quad \mathbf{let}\ (\operatorname{ran} \iota \cap \operatorname{dom} \omega) = \{c_1, ..., c_n\}\ \mathbf{in} \\
\quad\quad\quad \texttt{Any2OneChannel endManager = new Any2OneChannel();} \\
\quad\quad\quad \mathit{InitChanInfoMS}\ (\mathit{fst}\,\omega)\ P \\
\quad\quad\quad [\![\mathit{ProcessManagerMultiSync}(\mathit{Proc})\ || \\
\quad\quad\quad \left(\mathit{ControllersManager} \left(\begin{array}{l} \mathit{MultiSyncControl}(\mathit{from\_c}_1, \mathit{to\_c}_1) \\ ||\ ... \\ ||\ \mathit{MultiSyncControl}(\mathit{from\_c}_n, \mathit{to\_c}_n) \end{array} \right) \right)]\!]^{\mathit{Proc}} \\
\quad \mathbf{else}\ [\![\ \mathit{Proc}]\!]^{\mathit{Proc}}
\end{array}
$$

This function is used in the translation rules for parametrised and non-parametrised process definitions.

The class GeneralChannel

The class `GeneralChannel` was conceived to generalise and simplify the translation rules. In the original translation strategy, a channel that took part only in simple synchronisations was translated as a simple `Any2OneChannel`. Multi-synchronised channels, as explained before, are implemented with a pair of channels *to* and *from*, for communication with the controller. Several rules for declaration, instantiation and use of channels had different versions for the multi and simple synchronised cases.

In our strategy we decided to unify these rules: we define a class `GeneralChannel` that can represent both a multi-synchronised and a simple synchronised channel. It encapsulates the data and algorithms necessary for both cases. The set up is done in the constructor. The use of this class also makes simpler to determine if a synchronization channel will be used as a reader or a writer.

The class `GeneralChannel` defines constants *SIMPLE* and *MULTI* that determine if the channel is used in simple or multi-synchronisation, and *READ* and *WRITE*, that states if it is for reading or writing. Besides, it defines the following attributes:

- `private Any2OneChannel[] from`. The array *from* of the multi-synchronisation strategy. It is used only if the channel is multi-synchronised.
- `private Any2OneChannel to`. The channel *to* of the multi-synchronisation strategy. It is used as a regular channel in the simple synchronised case.
- `private String procName`. The name of the process that uses this instance of the channel.
- `private ChanInfo chanInfo`. A table that maps the name of the each process that uses the channel to an integer. In the simple synchronised case, the integer is zero if the channel is a writer, and 1 if it is a reader. In the multi-synchronised case, this is the identification of the process regarding the multi-synchronisation, that is, it represents the environment *MultiSyncEnv*.
- `private int rw`. This attribute determines if this instance is a reader or a writer.
- `private int sm`. This attribute determines if this channel is simple or multi-synchronised.

The class constructor takes the array `from`, the single channel `to`, the `chanInfo` and the name of the process. If the array is not null, then the channel is multi-synchronised and the attribute `sm` is set up with the constant `MULTI`; otherwise, it is assigned the value `SIMPLE`. It takes from the table `chanInfo` if the instance is a reader or a writer; recall that in the multi-synchronised case, the writer always has identification zero.

```
/**
 * Constructor for multi-synchronisation.
 */
public GeneralChannel (Any2OneChannel to, Any2OneChannel[] from,
        ChanInfo chanInfo, String procName) {
    ...
}


/**
 * Constructor for single-synchronisation.
 */
public GeneralChannel (Any2OneChannel[] from,
        ChanInfo chanInfo, String procName) {
    ...
}
```

The class also defines a constructor that builds a `GeneralChannel` from another one, changing only the `procName` attribute. This constructor is used in a call to a process so the status of reader or writer of the new instance can be changed accordingly. The new instance preserves the references to channels *to* and *from*.

```
/**
 * Constructs a channel from another one, changing only the
 * process id.
 */
public GeneralChannel (GeneralChannel gc, String newProcName) {
    gc.getTo(), gc.getFrom(), gc.getChanInfo(), newProcName);
}
```

Figure 3.5 shows the code for reading from a `GeneralChannel`. It tests the attribute `ms` to check if the channel is simple synchronised. If it is, it just calls the `read` method from channel `to`; otherwise, it executes the code for the multi-synchronised case. The code for `write` follows the same idea.

Figure 3.6 shows the method `synchronise`, which is called by synchronisation channels. It calls either `read` or `write`, depending on if this instance is a reader or a writer. The classes described in this section, namely, the class `GeneralChannel` and the classes for the multi-synchronisation protocol, are part of the utility library, *JCircusUtil*, which is provided with *JCircus* [16].

3.7 Generic channels and synchronisation on product values

Circus permits the declaration of channels that communicate or synchronise on tuples of values, in the same style as CSP. These types of channels are declared using the cartesian product operator.

```
public Object read() {
    Object r;
    if (this.ms == SINGLESYNC) {
        r = this.toController.read();
    } else {
        Vector seqOfSync = new Vector();
        Object[] sync = new Object[] {
            this.fromControler[this.procIdMS],
            this.toController,
            new Integer(this.procIdMS),
            new Integer(GeneralChannel.WRITER_ID)};
        seqOfSync.addElement(sync);
        MultiSyncClient client = new MultiSyncClient(
                seqOfSync, new Vector(), null);
        client.run();
        r = client.getValueTrans();
    }
    return r;
}
```

Figure 3.5: Implementation of method `read`

Let us take as an example the following channel.

channel $c : \mathit{Colour} \times \mathit{Colour} \times \mathit{Colour}$

Channel c communicates tuples with three components, each of a free type *Colour*; we say that c contains three fields. Some examples of possible uses of this channel are: $c.\mathit{blue}.\mathit{red}.\mathit{green}$, $c.\mathit{red}.\mathit{blue}?x$ and $c.\mathit{red}?x!\mathit{green}$.

In our strategy, for simplification, we consider that such types of channels can contain at most one communication (? or !) field, and it must be the last one. Therefore, the communications $c.\mathit{blue}.\mathit{red}.\mathit{green}$ and $c.\mathit{red}.\mathit{blue}?x$ are valid, whereas $c.\mathit{red}?x!\mathit{green}$ is not. They are translated as arrays of `GeneralChannel`s where each synchronisation field $.\mathit{exp}$, that is, each field that is not a communication field, is implemented as an additional dimension. Each value within the field type is represented by a position in this dimension.

Arrays are also used to translate generic channels. A generic channel communicates or synchronises on a value whose type is only determined at the moment of the use of the channel. For instance, the following declaration is taken from the case study of a Fire Control System in [32]:

channel $[T]\mathit{switchLamp} : T \times \mathit{OnOff}$

This channel is used to switch lamps on or off. The channel fields are the identifier of the lamp and a value of type $\mathit{OnOff} ::= \mathit{On} \mid \mathit{Off}$. A lamp can be of two types: area lamps, that are switched on when a fire is in an area, or fault lamps, used to indicate the occurrence of a problem in the system. To switch on the lamp that indicates that a fire has been detected in area 0, one must execute the communication $[\mathit{AreaId}]\mathit{switchLamp}.A0.\mathit{On}$, where $A0$ is an element of the free type *AreaId* that is used to instantiate the generic type.

Each generic type also adds a dimension to the array of channels. So, the above example is translated as `switchLamp[Type.AreaId][AreaId.A0][OnOff.On].synchronise()`. The first

```
public Object synchronise(Object x) {
    Object r = null;
    if (this.rw == GeneralChannel.READ)
        r = this.read();
    else
        this.write(x);
    return r;
}
```

Figure 3.6: Implementation of method `synchronise`

dimension is the instantiation of the generic type, and the second and third dimensions determines the synchronisation values. The indexes of the arrays are determined by constants defined in the `Type` class, and in the classes for each free type. From this strategy for implementation arises the restriction that only finite types can be used for fields and instantiation of generic channels; infinite types, such as $\mathbb{A}$, would lead to infinite arrays.

Another requirement is that a channel cannot be used as a communication channel and as a synchronisation channel in the same process. For instance, the following uses of channel *switchLamp* are not permitted within the same process:

$$[AreaId]switchLamp.A0.On \rightarrow \ldots \quad \text{(a)}$$

$$[AreaId]switchLamp.A0!On \rightarrow \ldots \quad \text{(b)}$$

The first use would generate an array with three dimensions, whereas the second would generate an array with two dimensions, because the last field is a communication, and not a synchronisation. The environment *SyncCommEnv* records, for each process, if a channel is used for communication or only for synchronisation.

3.8 Final considerations

The translation strategy introduced in this chapter is different from the original one, presented in [29]. The original translation strategy is presented in a didactic way: first, it presents rules for translation of programs that do not deal with multi-synchronisation and generic channels; then, the strategy is extended to handle these features, with the definition of new versions of rules for declaration and use of channels.

In the original strategy, a simple synchronised channel was implemented by the JCSP class `Any2OneChannel`. The implementation of multi-synchronised channels did not follow the principle of cohesion. Instead of being modeled as a single class, a multi-synchronised channel was represented with a pair of objects: an array of `Any2OneChannel`s, and one `Any2OneChannel`. This brought problems to define arrays of channels in the implementation of the kinds of channels described in Section 3.7. Because of that, the original strategy did not support the implementation of generic multi-synchronised channels or multi-synchronised channels on product values.

In our new proposal for the strategy, we came up with the class `GeneralChannel`, which can represent a simple or multi-synchronised channel. This design brought some advantages: it was possible to unify the different versions of the rules for channels; the treatment of multi-synchronisation was simplified, because the complexities of the protocol are encapsulated in the methods of `GeneralChannel`; and it was possible to implement multi-synchronised channels of the kind described in Section 3.7, because arrays of channels are now arrays of `GeneralChannel`s.

The translation strategy as presented in this thesis is the one implemented by *JCircus*; it is part of the documentation of the tool. The complete strategy can be found in Appendix A. In Chapter 5, we present a detailed comparison between the new translation strategy presented here, and the original strategy described in [29]. In the next chapter, we present *JCircus*.

Chapter 4

JCircus

This chapter describes *JCircus*, the translator to Java that we implemented. *JCircus* receives as input a *Circus* specification written in LaTeX. It parses and typechecks the specification; if no errors are found, then it applies the translation strategy described in the last chapter, generating a Java program that implements the specification.

The *Circus* parser [7] and the *Circus* typechecker [45] were contributions from colleagues from the *Circus* group. The main contribution of the work described in this thesis is the module that performs the translation to Java. *JCircus* was implemented using the CZT [1] framework, and runs under Microsoft Windows.

The project was documented in the Unified Modeling Language (UML), version 1.3 [8]. In Section 4.1, we present a report on the requirement analysis – the requirements of the tool, illustrated with Use Case Diagrams, and the requirements on input specifications. Section 4.2 discusses design and implementation issues. Section 4.3 shows how to use *JCircus*, with an example of translation. Section 4.4 presents the test strategy that we followed to test *JCircus*.

4.1 Requirements

JCircus is a tool with simple functionality: its goal is to generate a Java program that implements a *Circus* specification. Basically, interaction with the user is required only for entering the input specification and the parameters for the translation. Error messages are also shown, when necessary.

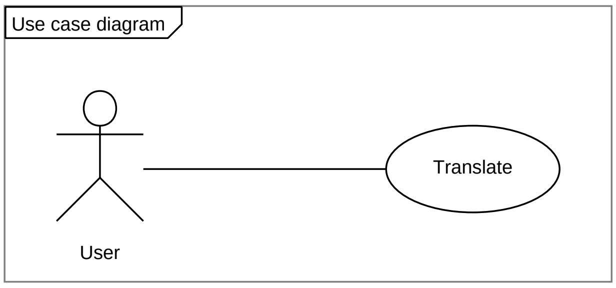
Not any *Circus* program is possible to be translated. The programs must comply to a number of pre-requisites that are justified by the restrictions on the JCSP library, and on the solutions found to implement some features like multi-synchronisation, for instance.

In what follows, we specify the interaction with the user and list and comment on the restrictions on input programs.

4.1.1 Requirements of the tool

JCircus should provide a friendly interface with the user. We present the operation of the tool in the use case in Figure 4.1. The use case diagram is a figure that represents the functionalities of the system. In the diagram, actors are represented by stick figures and the use cases are represented by ovals. In our system, the actor is the user, who can ask to translate a *Circus* program.

A *Circus* specification can contain several process definitions. The user is required to determine for which of these he or she wants a main class to be defined. For the chosen processes, the system



Use Case: Translate
1. The user enters the path of input specification, the project name and the output directory.
2. The system checks if the input specification is well-formed, well-typed and meets the requirements.
3. If there is some problem, the system gives an error message explaining the problem.
4. If not, the system asks for which process definitions the user wants to create a main class.
5. For each process entered by the user, the system checks if it requires parameters. If it does, the user is asked to enter their values.
6. The system parses and typechecks the values entered. If some problem is found, an error message is given.
7. If no problem is found, the program is translated and the files are created in the output directory.

Figure 4.1: Use case diagram and description

creates a class called `Main_<Proc>` (where `<Proc>` is the process name), and a simple graphical interface, which will be used to run the program that results from the translation. The graphical interface is not formalised by the translation rules and was an additional functionality that we implemented; it will be presented in Section 4.3.

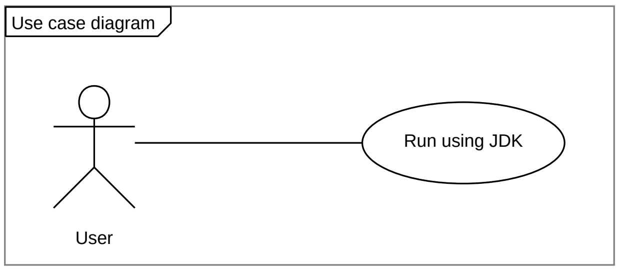
A process definition can introduce a non-parametrised process, like the ones in the GCD example, or a process that takes parameters. If a parametrised process is chosen as a main process (Rule A.3), the user needs to determine the actual values that he or she wants to pass to the parameters. These will be passed in the constructor of the class that implements the process, which is called in the class `Main_<Proc>`. Like the input specification, the parameters are also entered as a LaTeX string.

The output of the translation is Java code composed of several files and allocated in packages; these will form a Java project. The user must determine a name for the project `<proj>`, and the output directory where it is to be created `<dir>`. If not already existent, a folder `<dir>\<proj>\src\<proj>` will be created; and inside this folder, the project folders `axiomaticDefinitions`, `gui`, `processes` and `typing` will be created and they will contain the source code for the project.

The requirements of the translator are captured by the laws of the translation strategy, which were discussed in the previous chapter. They are all presented in Appendix A.

To run the project, the user needs to have a Java compiler and a Java Virtual Machine installed in the computer. The system generates a batch (.bat) file for each process definition that have a main class. The batch file compiles the project and executes the main class using JDK (version 1.3 and above) [2]. Alternatively, the user can use a IDE for Java like Eclipse or Netbeans, to create a project importing the source files. To compile and run the project, some auxiliary classes are necessary, for instance, the class `GeneralChannel` and the classes for implementation of the multi-synchronisation protocol; they are all grouped in the file `CircusUtil.jar`[1] , and are

[1]The Java Archive (.jar) is a file format that makes it possible to bundle multiple files into a single archive file.



Use Case: Run class `Main_<Proc>`
1. Execute the .bat file `run_Main_<Proc>.bat` located in `<dir>\<proj>\src`, passing as argument the path of `JCircusUtil.jar`.

Figure 4.2: Use case diagram

provided with *JCircus* [16]. Figure 4.2 depicts the use case diagram for running a process using the JDK platform.

4.1.2 Requirements on input specifications

Only concrete *Circus* programs can be translated. As said before, a concrete *Circus* program is a program that does not use schemas and specification statements in action definitions, and all guards are conditions. An abstract specification must be refined prior to the application of the translation strategy. Other restrictions are syntactic and can be enforced by a pre-processing of the input; they are listed below.

- *The only types supported are free types and $\mathbb{A}$.*
  The type $\mathbb{A}$ is a given type defined in the *prelude* section of Z and represents a general number type. Naturals and integers have type $\mathbb{A}$ in the Z/*Circus* type system. Free types are types defined by the writer of the specification and is also a basic type in the Z/*Circus* type system.

  A restricted set of operators can be used to build compound types, for example, types that represent sets of sets, cartesian products and schemas. The representation of such kinds of types is not trivial and for that, we have decided to handle only the basic types. A more detailed discussion about the implementation of types is presented in Chapter 5.

- *Z paragraphs are axiomatic definitions of the form $v : T \mid v = e$, free types, or schemas of the form $[x_{11}, \ldots, x_{1n} : T_1; \ \ldots; \ x_{n1}, \ldots, x_{nn} : T_n \mid inv]$ used only to define the state of a basic process.*
  The translation of such constructs has been commented in Chapter 2. The predicate part *inv* in the schema is ignored since it has already been considered in the refinement process.

- *A variable name cannot be redeclared in a nested variable block.*
  Variable blocks in *Circus* are translated as variable blocks in Java. Differently from *Circus*, Java does not allow redeclaration of variables in inner scopes. Thus, the translation of *Circus* programs with redeclaration of variables would result in a Java program that does not compile. Other scope rules, for instance, for method parameters and state elements are compatible between *Circus* and the Java translation.

- *Hiding of channels appears only as the last operator applied to a process in a process definition.*
  This requirement is justified by the way in which hidden channels are instantiated. Hidden channels are initialised in the constructor of the process in which they are hidden. This means that they are visible in any sub-process or action inside this process definition. Therefore, if a channel were hidden in a sub-process or action the implementation would be inaccurate, because it would not be actually hidden from the other actions and processes.

- *Only prefixing actions, guarded or not, can be branches of an external choice.*
  The *Circus* external choice operator is implemented with the `Alternative` class of JCSP. Its constructor takes the list of channels that take part in the choice. In order to make easy the identification of the initial channels of a choice, we restrict the format of actions in an external choice to be only prefixing actions, guarded or not.

- *The communication in the prefixing action in an external choice cannot be an output communication.*
  This restriction follows a limitation of JCSP. All channels that take part in the `Alternative` must be an instance of the abstract class `AltingChannelInput`, which is an input channel.

- *The synchronisation sets in any parallel composition are the intersection of the sets of channels used by the parallel actions or processes. Interleaved actions or processes must not have any channel in common.*
  The JCSP parallel construct does not allow the definition of a synchronisation channel set; processes will synchronise on all channels that are common to the parallel processes or actions. For this reason, the intersection of the alphabets determines this set: if it is not empty, we have a parallelism; otherwise, we have actually an interleaving.

- *Only free types can be used to define types of synchronisation fields in channels and to instantiate generic channels.*
  This requirement arises from the implementation of these types of channels, which uses arrays of `GeneralChannel`s. The use of the infinite type $\mathbb{A}$ would result in infinite arrays.

- *Only free types are used as the indexing set of iterated operators.*
  The reason is similar to the one for the previous requirement. The use of the infinite type $\mathbb{A}$ would lead to an infinite indexed set.

- *A multi-synchronisation must not define more than one writer.*
  The multi-synchronisation protocol requires that each process have an identification number regarding each of the multi-synchronisations on which it takes part and the writer must be the process with identification zero.

- *Multi-synchronisations on a channel must always involve the same processes.*
  As explained in the previous chapter, there is one controller for each multi-synchronised channel. The array of channels *from* is used for communication between the controller and the processes that want to engage in multi-synchronisation. This array is set up in the constructor of the controller, and cannot be modified afterwards. Therefore, the controller always interacts with the same process.

- *Communications that contains multiple fields can contain only one input or output field, and it must appear in the last position.*
  This is for simplicity of implementation. As discussed in the previous section, these types of channels are implemented using arrays. The values of the synchronisation fields are positions

in the array. The last field is the value actually communicated, if it is an input or output field.

Some of these requirements are not serious restrictions because they can be achieved with refactoring. This is the case of the requirements about the redeclaration of variables and the hiding operator. Variables can be renamed, and new processes can be defined to substitute a process with hidden channels in a compound process.

Other requirements impose restrictions on syntactic constructs that are more often and cannot be easily avoided. The support for more types in the specification, the flexibility in formats of actions that participate in external choices, and the possibility of defining the synchronisation sets in a parallelism are some examples of features that we would like to implement. Investigations on how this could be done are interesting pieces of future work.

4.2 Design and implementation

JCircus was implemented in Java, and was constructed using the CZT framework [1, 25], which is an open-source Java framework for the ISO Standard Z [33] and its extensions. The framework provides, among other things, a Java library for abstract syntax trees, basic tools like parsers, type checkers and printers, and an interchange format, base on XML, for representing specifications. We have chosen CZT to use as a basis for our tool because of the quality of its design and the resources that it offers, which facilitate code reuse and maintenance.

In Section 4.2.1, we describe the object representation for a *Circus* specification within the framework. This is the heart of the system, since all operations carried out in the specification (namely, parsing, typechecking and translation) involve the manipulation of this representation. Section 4.2.2 presents an overview of the *JCircus*' architecture, with the modules that compose the software. In Section 4.2.3, it is shown how the environments from the translation strategy are implemented in *JCircus*. The last section gives more details about the translator module of *JCircus*.

4.2.1 The *Circus* AST

AST is an acronym for Abstract Syntactic Tree; this is an object representation of a parsed *Circus* specification. The CZT framework defines interfaces and default implementations for the elements of an AST; it has recently been extended to include classes and interfaces for a *Circus* AST. For instance, a *Circus* action definition is represented by the interface `ActionPara`; the default implementation provided within the framework is `ActionParaImpl`, which defines two attributes: the name of the action (`DeclName`) and the *Circus* action (`CircusAction`).

Figure 4.3 shows part of the inheritance hierarchy of the *Circus* AST. All nodes extend a common interface `TermA` - a node to which a list of annotations can be added. An annotation can be any kind of Java object, and it contains relevant information about that node. Typical examples of annotations are `LocAnn`, which registers the line and column where the construct appears in the source file; and `TypeAnn`, added by the type checker to all expressions of a well-typed specification.

The inheritance hierarchy of the Java interfaces have some correspondence with the structure of the *Circus* syntax. For instance, the `Para` interface represents a paragraph of any kind. The interfaces `AxPara`, `ProcessPara` and `ActionPara` extend `Para`; they represent axiomatic definitions, processes declarations and action definitions, respectively, which are all paragraphs. The leafs are concrete classes that can be instantiated, while the inner nodes are abstract classes.

The use of an AST allows easy access to syntactical constructs from within the Java program. The attributes can be accessed using the getter methods defined in the interfaces. It is also

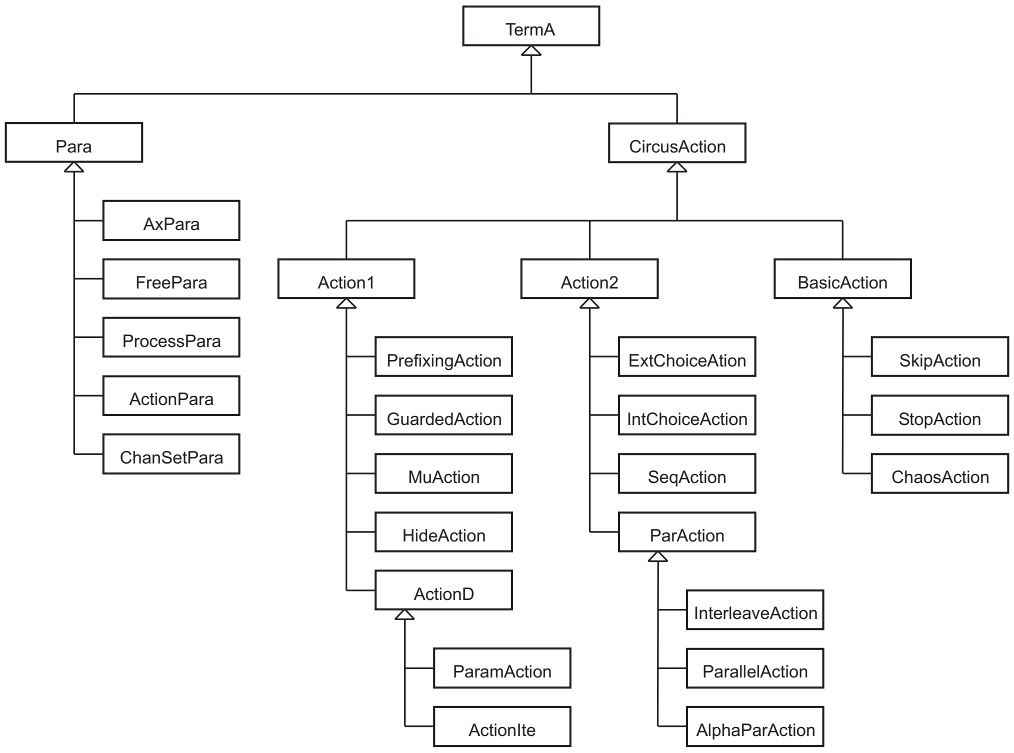


Figure 4.3: *Circus* AST inheritance hierarchy

possible to access the tree in a systematic way using the implementation of the CZT visitor design pattern [25], which is a variant of the *visitor design pattern* proposed by Gamma et al [20].

The CZT visitor design pattern

The *visitor design pattern* [20] is a design solution that provides a way to separate an algorithm from an object structure. The result of this separation is the ability to add new operations to existing object structures without modifying those structures.

In the standard version of this pattern [20], this is accomplished with a double dispatching mechanism. An interface, the `Visitor`, defines `visit` methods for each class of object that is to be visited. The classes, in turn, must provide an `accept(Visitor v)` method, which calls back the correct visit method for its class. The disadvantage of this approach is that it makes it difficult to extend the set of classes to be visited (in our case, the AST classes), because each new class requires the definition of a new method in the `Visitor` interface, which in turn requires modification in all its implementations. Another disadvantage is that all visitors are required to implement a method for each class, even the ones that are irrelevant for the operation that the visitor performs.

The *CZT visitor* design pattern [25] is a combination of two variations of the standard visitor pattern: the *acyclic visitor* pattern [26] and the *default visitor* pattern [23], and incorporates their advantages. The *acyclic visitor* pattern allows that new AST classes can be added without changing the existing visitor classes, which means that a visiting operation does not have to be defined to all AST classes. The *default visitor* design pattern takes advantage of the inheritance relationships to make possible the implementation of default behaviour for nodes that have a common superclass. We will show how these characteristics facilitate the implementation of the two visitors implemented in our tool.

4.2.2 *JCircus* architecture

JCircus is composed of three main modules. The first is a parser, which receives a LaTeX file containing the specification, parses it, and creates the AST that represents the specification. The AST is given as input to the type checker, which performs type inference, checks for type errors, and annotates the AST nodes for expressions with their types. Figure 4.4 shows the modules of *JCircus*. The *Circus* parser [7] and *Circus* type checker [45] are contributions of colleagues in the *Circus* group, and the *Circus* AST is part of the CZT project. The Translator module is the main contribution of this thesis. The *JCircusGui* component is also new to this thesis, but will not be detailed here, as it is just a graphical interface for the tool.

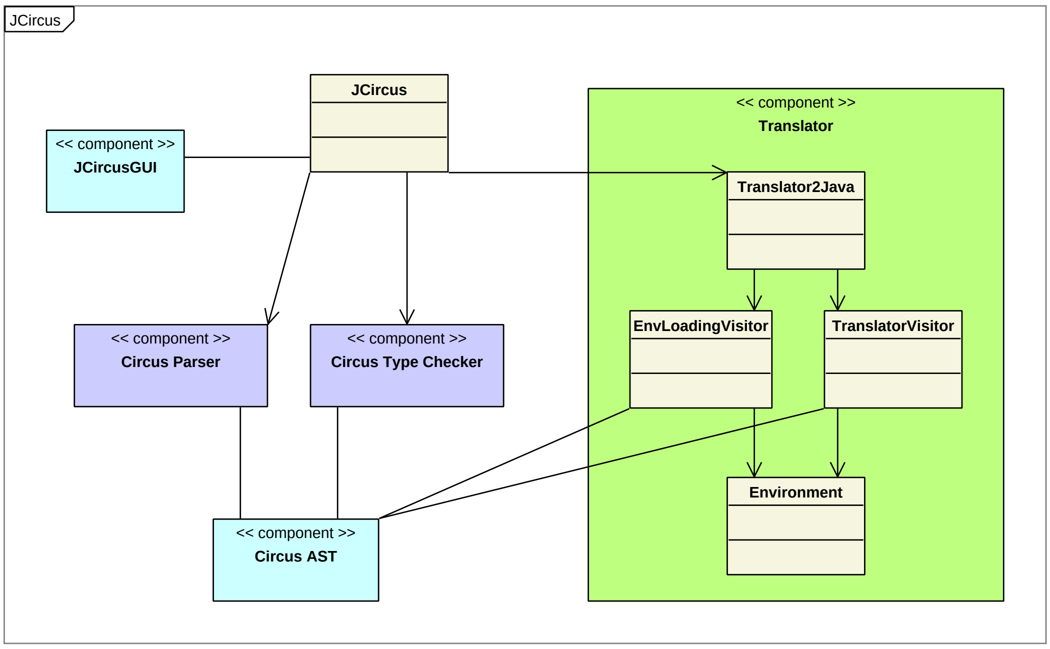


Figure 4.4: *JCircus* architecture

The annotated syntax tree is input to the translator module. The translator module executes the translation in two phases: the first is a pre-processing of the AST tree, and the second is the generation of Java code. After that, it creates the source files for the project.

In the pre-processing phase, *JCircus* loads environments containing information about channels and types, and the AST is annotated with relevant information to be used in the second phase of the translation. The environments correspond to the environments introduced in Chapter 2, but some adaptations were necessary on the way the data is represented. These adaptations will be explained in the next section.

In the second phase, the translation rules are applied to generate the Java code. This phase of the translation uses the information on environments and the annotations on the AST. Both operations (pre-processing and translation) are implemented using the CZT visitor design pattern. In the following sections, we explain details of these operations.

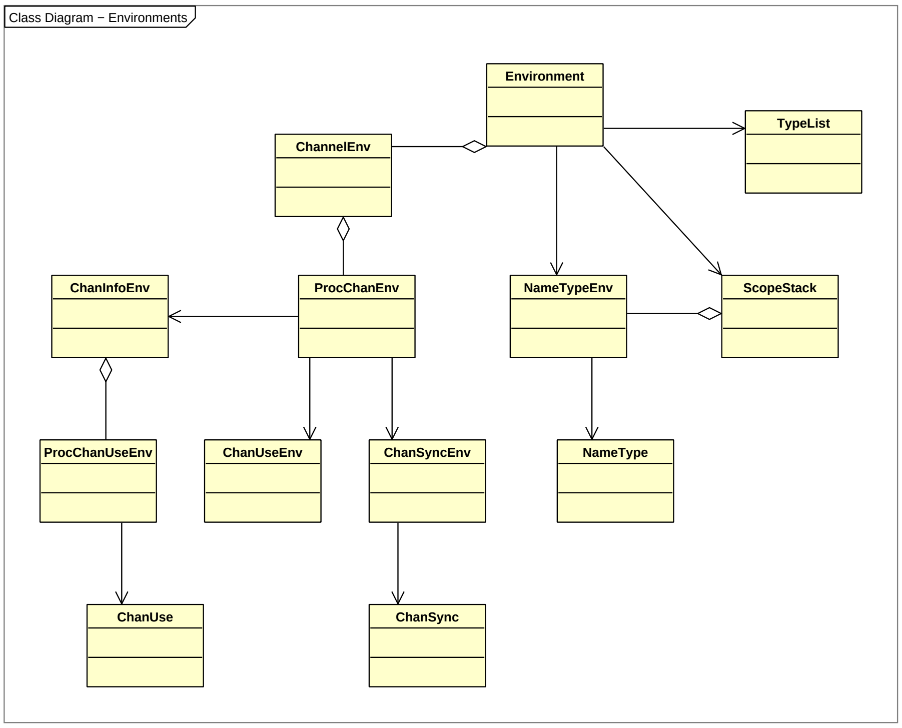


Figure 4.5: Class diagram for environments

4.2.3 Environments

In our implementation of the translation strategy, not all the environments defined in the specification of the strategy needed to have an explicit corresponding component in the design. The environment *ChanTypeEnv*, which maps channel names to their *Circus* type was not necessary, because the type checker annotates the AST nodes for expressions and channels with their types. Therefore, to find out the type of a channel, we can simply get the annotation in the node. Figure 4.5 presents a class diagram for the environment-related classes.

The environments *VisChanEnv* and *HidChanEnv*, which hold a sequence of visible and hidden channels of a process, are implemented by instances of the class `ChanUseEnv`. The environment *SyncCommEnv*, which maps each channel name into S or C, is implemented by the class `ChannelSyncEnv`. It maps the channel name to an enumerated type `ChanSync`, which can assume the values `S` or `C`. An element is added to this environment in the visit of a communication; an auxiliary function is called to find out the type of the channel regarding synchronisation. For example, the instance of the environment `ChannelSyncEnv` the process $GCDEuclides_1$, contains the following mapping: $in \mapsto C$, $out \mapsto C$, because both are communication channels, that is, they contain an input or output field.

Information about how channels are used within the sub-processes of a compound process is registered in `ChanInfoEnv`. In the strategy, this class corresponds to environments *ReadWriteEnv* and *MultiSyncEnv*; both are used to initialise the component `ChanInfo` (see the definition of functions *InitChanInfoMS* and *InitChanInfoSS* in Rule A.3). The environment `ChanInfoEnv` maps a channel name into an instance of `ProcChanUseEnv`, which, in turn, maps a process name into an object of type `ChanUse` (`Input`, `Output` or `Undefined`).

When a channel is multi-synchronised, the `ChanUse` of the channel in each parallel process is used to determine its multi-synchronisation identification number. The following example shows how this is done.

$$\textbf{process}\ A \mathrel{\widehat=} \textbf{begin}\ \bullet\ ch!1 \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ B \mathrel{\widehat=} \textbf{begin}\ \bullet\ ch?x \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ C \mathrel{\widehat=} \textbf{begin}\ \bullet\ ch?x \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ ParABC \mathrel{\widehat=} (A \mathbin{|[} \{\!| ch |\!\} \mathbin{]|} B \mathbin{|[} \{\!| ch |\!\} \mathbin{]|} C) \setminus \{\!| ch |\!\}$$

The environment `ProcChanUseEnv` for channel *ch* and process *ParABC* contains this mapping:

$$\{A \mapsto \mathit{Output}, B \mapsto \mathit{Input}, C \mapsto \mathit{Input}\}$$

As explained in Chapter 3, the process *ParABC* is translated into a Java class `ParABC`. The channel *ch*, which is hidden, is instantiated within the constructor of this class (see Rule A.3, function *HiddenCCreation*). The instantiation of the channel, requires an object of type `ChanInfo`, which is a mapping of a process name into an integer (see Section 3.6). In the multi-synchronised case, this is the multi-synchronisation identification number. The generation of code for the instantiation of `ChanInfo` uses the environment `ProcChanUseEnv` to determine the mapping in `ChanInfo`. For process *ParABC*, this mapping is used to determine which process will be the writer; in this case, it is process A, as it is the only that is mapped to *Output*. If there are two processes mapped to *Output*, the tool gives an error message, since this breaks the requirement that there must be at most one writer per parallelism.

The environments `ChanUseEnv` (hidden and visible), `ChanSyncEnv` and `ChanInfoEnv` are defined for each process. They are grouped in the class `ProcChanEnv`. An environment for channel information in each process (`ChannelEnv`) maps a process name to an instance of `ProcChanEnv`. For instance, in our GCD example, the channel environment `ChannelEnv` contains three entries, one for each process ($GCDEuclides_1$, *GCDClient* and *SumOrGCD*); each of these map the name of the process to an object of type `ProcChanEnv`, which contains the correspondent environments `ChanUseEnv` (hidden and visible), `ChanSyncEnv` and `ChanInfoEnv`, defined for each process.

The *TypesEnv* environment lists all types that are used within the *Circus* program being translated. This information is used to define constants for each type in the generated class `Type`. This environment is implemented as an instance of `TypeList`, which contains a list of the names of all free types and the number type. In our GCD example, there is not any free type definition; so, the type environment contains only one reference to the built-in type *arithmos*. In our implementation, types are represented by the class `CircusType`.

The environments *StateCompEnv* and *LocalVarEnv*, used in the translations of recursive and parallel actions, are implemented by class `NameTypeEnv`, which maps a name to its classification. This environment is loaded during the pre-processing phase and is added as annotation to all nodes that represent parallel or recursive actions. It contains all the variables that are in scope at the moment of the the recursive or parallel action is used. During the translation phase, the annotation is retrieved, and the information in it is used to declare the copies of the variables. The `NameType` information permits to distinguish state components from local variables.

All the classes for environments are encapsulated in a class `Environment`. The visitors have access to an instance of this class and from it they have access to all the environments discussed above. The classes `ScopeStack` and `NameTypeEnv` are used in the pre-processing phase only; they are explained in the next section.

4.2.4 The translator module

The translator is the main component of *JCircus*; it is the module that actually implements the translation strategy. As mentioned before, the first phase of the translation is a pre-processing of the specification, and the second is the generation of Java code. Details of these operations are given in the next sections.

Pre-processing of the specification

The pre-processing is implemented by the class `EnvLoadingVisitor`. The starting point of this visitor operation is a `Spec` node, from which all nodes of the AST can be accessed.

The pre-processing is responsible for loading the environments described in the last section, and annotating the AST with `NameType` information. The `NameType` annotation classifies a reference expression (an identifier used as an expression). There are seven reference types which are summarised in Table 4.1. This information is fundamental to the translator because a *Circus* name can be translated into different forms of Java constructs. We exemplify this with the following assignment command:

$$x := a$$

This *Circus* assignment is translated into a Java assignment (Rule A.51). The function *JExp* translates a reference expression according to the reference type (see the definition of the function in Appendix A). The variable x could be a state component, a local variable or a process or action parameter; in any case, it is translated into a Java variable or attribute. The name a, however, can also be a constant, defined in an axiomatic definition or in a free type. In this case, it is translated into a method or constructor call. Thus, the translation of this assignment could be one of these four possibilities:`x = a`, if a is a parameter, state component or local variable; `x = a()`, if a is an axiomatic definition local to the basic process where the action is defined; `x = AxiomaticDefinitions.a()`, if a is a constant defined in a global axiomatic definition; or `x = new FT(FT.a)`, if a is an element of a free type FT.

This information is not provided by the parser nor the type checker; In any case, the syntactic category of a is the same, and the type annotation cannot be used to differentiate between variables and constants that have the same type. To get this information, we would have to know all the scopes of declaration of variables. However, when the type checker finishes its task, only the global environment is available.

To collect this information we need to identify the scopes in a similar way that the type checker does to find out the types of names previously declared. We define an environment represented by class `NameTypeEnv`, which associates a name to its `NameType`. The environment contains one instance of this class, which represents the current scope. Another class, `ScopeStack`, is a stack of `NameTypeEnv` which represents the scopes that are overridden when a new declaration occurs: when declaration statement is found, the current environment is pushed into the stack; when its scope finishes, the environment on top of the stack is retrieved.

The environment contains one instance of `NameType`, which represents the current scope; and one instance of `ScopeStack`. While traversing the AST, it is possible to know where the declaration occurs - if as a process parameter, action parameter and so on - because we know in which kind of AST node we are. The name is inserted in the environment with its correspondent `NameType`. When a reference is found, the `NameType` is retrieved from the current environment and used to annotated the AST node for the reference. When the pre-processing finishes, all references have been annotated with their `NameType`s and this information can be accessed in the subsequent phase, the generation of Java code.

Name Type	**Example of declaration**	**Translation of x := a**
Free type element	$FT ::= a \mid b \mid c$	`x = new FT(FT.a)`
Global constant	$a : \mathbb{N} \mid a := 1$ (declared in global scope)	`x = new AxiomaticDefinitions.a()`
Local constant	$a : \mathbb{N} \mid a := 1$ (declared in a local scope)	`x = this.a()`
Process parameter	$P \mathrel{\widehat=} a : \mathbb{N} \bullet \mathbf{begin} \ldots$	`x = a`
Action parameter	$A \mathrel{\widehat=} a : \mathbb{N} \bullet \ldots$	`x = a`
State component	$S == [a : \mathbb{N} \bullet \ldots]$	`x = a`
Local variable	$\mathbf{var}\ a : \mathbb{N} \bullet \ldots$	`x = a`

Table 4.1: Translation of reference expressions

The implementation of this visitor provides evidence of the advantage that the CZT visitor brings from the *default visitor* design pattern: the possibility of allowing default implementations for classes that extend from a common superclass. The `EnvLoadingVisitor` is mainly concerned with collecting information about channels, types, and references. Binary actions, for instance, are mostly visited in the same way: each action is visited and then their environments are combined to form the environment for the binary action. This implementation is provided in the visiting method for `Action2`, which is called for all binary actions that do not provide an specific implementation. This minimises code duplication: if the standard visitor were being used instead, we would have to define identical visiting methods for all subclasses of `Action2`.

Generation of Java code

The translation is basically the application of each translation rule in Appendix A to each node of the abstract syntax tree. This is accomplished by the visitor `TranslatorVisitor`. This visitor is called for each process definition and free type definition; for each of them the visit method returns the Java code resulting from the translation.

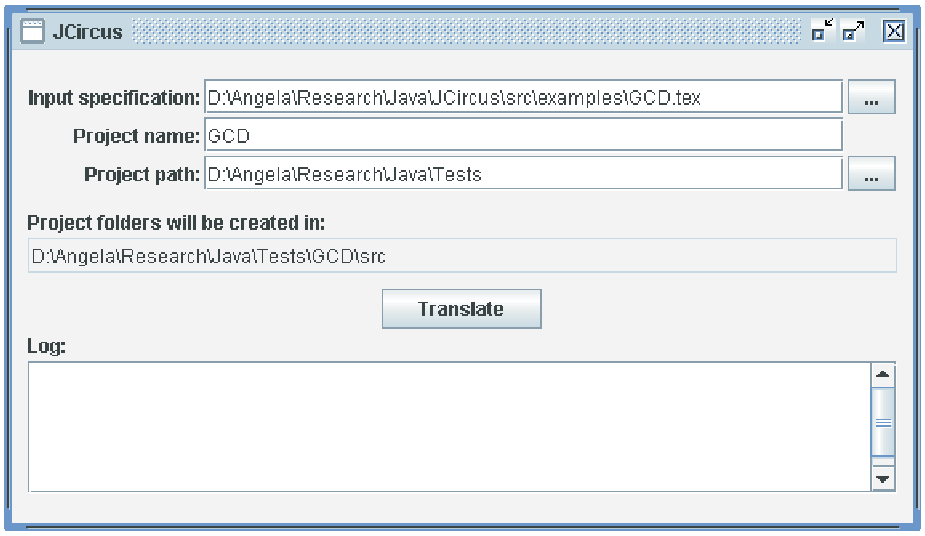
We used the Velocity engine [4] for generation of the Java code. It permits that part of the Java code to be generated is written in a separate file (a template), instead of directly in the visitor methods. This facilitates code maintenance.

The implementation `TranslatorVisitor` shows the advantages that the CZT visitor incorporates from the *acyclic visitor* design pattern. The `TranslatorVisitor` deals only with syntactic constructs below the level of process declarations in the syntax; the nodes that represent axiomatic definitions and channel sets, for instance, need not to be visited, because they are not translated into Java code. The standard visitor design pattern would require the definition of visiting methods for all these constructs.

4.3 Using *JCircus*

In this section we will show an example of use of *JCircus*, for the GCD program introduced in Chapter 2. The initial screen of *JCircus* is shown in Figure 4.6. In this screen, the user enters the parameters for the translation, as explained in Section 4.1.1: the path of the input specification, the project name, and the project path. The screen shows where the project folders will be created; it is always inside a folder `src` in the project path.

After entering the parameters, the user can press the button Translate. If any of the parameter fields is empty, the tool gives an error message; otherwise, it parses, type checks, and translates the input specification. The text area Log shows error and success messages.


JCircus

Input specification: D:\Angela\Research\Java\JCircus\src\examples\GCD.tex

Project name: GCD

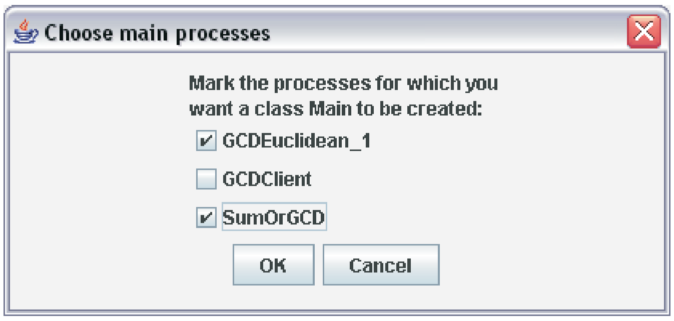
Project path: D:\Angela\Research\Java\Tests

Project folders will be created in:

D:\Angela\Research\Java\Tests\GCD\src

Figure 4.6: *JCircus* graphical interface

Our specification in file `GCD.tex` contains three process definitions. After parsing and type-checking the input file, *JCircus* identifies the process definitions, and shows a window (Figure 4.7) where we can select one or more processes for which we want a class main to be created. We can choose, for example, the processes $GCDEuclides_1$ and $SumOrGCD$. If any of these processes were parametrised, *JCircus* would ask the user to enter the parameters; as this is not the case, it proceeds and generates the source files successfully.


Choose main processes

Mark the processes for which you want a class Main to be created:

- [✓] GCDEuclidean_1
- [ ] GCDClient
- [✓] SumOrGCD

Figure 4.7: Window to choose the main processes

Besides the project packages defined in Section 4.1.1, *JCircus* also creates a package `<proj>.gui` (where `<proj>` is the project name). This package contains the source files for a graphical interface that is created to run with each process chosen to have a main class. For each of these processes, *JCircus* creates the main class, a graphical interface that represents the environment, and a `.bat` file. For instance, for process $GCDEuclides_1$, *JCircus* creates files `Main_GCDEuclides_1.java`, `Gui_GCDEuclides_1.java` and `Run_GCDEuclides_1.bat`. The `.bat` file can be used to compile and run the program using JDK (version 1.3 and above) [2].

The graphical interface for process $GCDEuclides_1$, in file `Gui_GCDEuclides_1.java`, defines the class `Gui_GCDEuclides_1`. This class represents the environment that interacts with the process, and is also an implementation of the interface `CSProcess`. It is also a Java Swing frame that runs in parallel with the process in the main class; it contains components that represent the events that the environment executes. The interface of a process with the environment is composed by only the channels that it uses and are not hidden. In the graphical interface, these are represented as buttons. The internal state, hidden channels and internal operations cannot be seen by the environment.

Figure 4.8 shows the graphical interface that is generated for process $GCDEuclides_1$. The process $GCDEuclides_1$ uses only two channels: the input channel *in* and the output channel *out*, and both communicate natural numbers. The communication fields are the text fields next to the buttons, where we can type values for the input channels or visualise the parameters of the output channels.

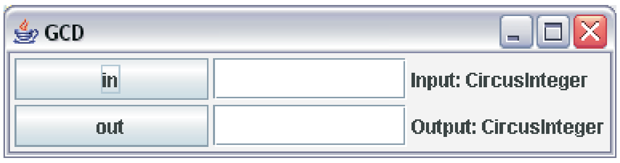


Figure 4.8: GUI for process $GCDEuclides_1$

The class main for a process also instantiates any channel that it uses and is not hidden, in the same way that it is done for hidden channels in the constructor of a class. Figure 4.9 shows the code for the main class of process $GCDEuclides_1$. It contains only one method `main`, which instantiates channels `in` and `out` and instantiates a parallelism between the process $GCDEuclides_1$ and the graphical interface. Both processes, $GCDEuclides_1$ and the graphical interface, use channels `in` and `out`. Process $GCDEuclides_1$ reads from channel `in`, so it is mapped to 1 in `ChanInfo`; the GUI writes on channel `in`, so it is mapped to zero. For channel `out`, it is the other way around.

When we run the class `Main_GCDEuclides_1`, the screen presented in Figure 4.8 is shown. The program waits for a synchronisation on channel *in*, as this is the first action that the process determines. As this is an input channel, we must type in the first text field the parameter, which is the first of the pair of numbers for which we want to calculate the GCD. After entering the parameter, we press the button `in`; this act represents the synchronisation on channel *in*, with communication of the value that has been entered in the text field. The generated program does not perform parsing or type checking. It relies that the values entered by the user are well-formed and well-typed. If this is not the case, an error will occur. Further work include an integration of the generated program with a parser and typechecker for expressions.

Once the first number has been entered, the program waits for the second synchronisation on channel *in*, that communicates the second number. After that, the program calculates the GCD and waits for synchronisation on channel *out*. When we press the button `out`, the GCD appears in the text field next to it.

Figure 4.10 shows the graphical interface for the process *SumOrGCD*. Interaction occurs in the same way as described above. This process uses channels *in* and *out*, but they are hidden from the environment, and for this reason, they do not appear in the graphical interface. Channels *sum* and *gcd* are only synchronisation channels. As they do not communicate values, they do not have a text field associated.

When using the interface generated automatically by *JCircus*, the user must be careful to only press a button if the program is waiting for synchronisation on the respective channel; otherwise, the program deadlocks. It would be interesting to have some mechanism that would enable or disable the buttons, to prevent the user from unintentionally pressing the wrong button. This would also help the user to visualise the flow of execution of the program. We can use an approach similar to that of the component `ActiveButtonControl` of the JCSP library. This class is part

```
public class Main_GCDEuclides_1 {

    public static void main(String args[]) {

        Any2OneChannel to_in = new Any2OneChannel();
        ChanInfo chanInfo_in = new ChanInfo();
        chanInfo_in.put("GUI", new Integer(0));
        chanInfo_in.put("GCDEuclides_1", new Integer(1));
        GeneralChannel in = new GeneralChannel(to_in, chanInfo_in,
                "GCDEuclides_1");

        Any2OneChannel to_out = new Any2OneChannel();
        ChanInfo chanInfo_out = new ChanInfo();
        chanInfo_out.put("GCDEuclides_1", new Integer(0));
        chanInfo_out.put("GUI", new Integer(1));
        GeneralChannel out = new GeneralChannel(to_out, chanInfo_out,
                "GCDEuclides_1");

        new Parallel(new CSProcess[] {
            new GCDEuclides_1(in, out),
            new Gui_GCDEuclides_1(
                    new GeneralChannel(in, "GUI"),
                    new GeneralChannel(out, "GUI"))
        }).run();
    }
}
```

Figure 4.9: Code for class `Main_GCD`

of the `jcsp.awt` package, which provides CSP extensions for all `java.awt` components. It is a user-programmable finite state machine for controlling an array of buttons, which can be used to enable or disable buttons. This simple extension of *JCircus* is left as future work.

In this version of the tool, we have decided that only the channels that do not take part in any communication within the process interact with the GUI. For instance, channels *in* and *out* in our example *SumOrGCD* would not appear in the GUI, even if they were not hidden, because they are involved in a synchronisation in the parallelism of processes $GCDEuclides_1$ and *GCDClient*. We have decided to do so to avoid the possibility of multi-synchronisation involving the GUI. We plan to generalise the approach to allow multi-synchronisation involving the GUI. The extension should be straightforward since the complexities involving multi-synchronisation are all concentrated in the `GeneralChannel` class.

The generation of the class main as described here and the graphical interface was an additional functionality provided by *JCircus* and is not formalised by the translation rules. These classes make the execution of the program generated with a simple interface immediately available. They are appropriate for the rapid prototyping of *Circus* programs. The classes that capture the behaviour of each of the processes, however, can be used in other contexts, where, for example, an interface that is more specific to the application is implemented.

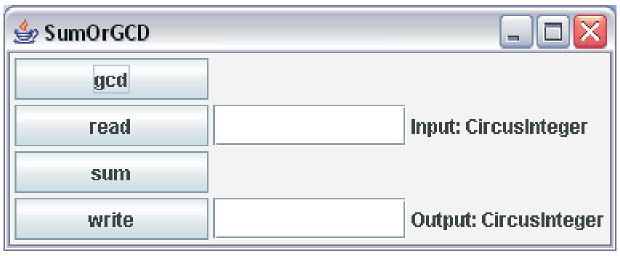


Figure 4.10: GUI for process *SumOrGCD*

4.4 Test strategy

The objective of our project was to construct a tool that implements the translation strategy and can translate *Circus* programs that comply with a list of pre-requisites. The development of our test strategy followed a structured approach so that we could end up with a system in which we have confidence.

JUnit [3] was used to carry out unit and regression testing. JUnit is a framework for automation of the execution of test cases in systems developed in Java. It allows developers to save time and effort in the process of testing.

In this section we describe the types of tests that were carried out, which aspects of the system they were aiming to test, the techniques used, and the problems that such tests managed to identify.

Unit testing

The objective of unit testing is to test the correctness of the implementation of a particular module of source code. We used unit testing to test the classes for environments and utility methods. JUnit was used to write assertions that compared the returned result of a function with the expected result for a number of test cases.

Function testing

We used function testing to test the correctness of the implementation of the translation rules. After the implementation of each rule, a simple *Circus* program was designed to test the application of that rule. Then, more elaborated test cases, involving the application of many rules were designed to produce more complex scenarios and test the validity of the rule on those scenarios. These tests revealed the errors in the rules for compound processes and action parallelism explained in the next chapter.

Integration testing

The parser is a very basic and important component of the system. It transforms the textual input file into an object representation of the program's syntax tree. The integration of a parser to the translator showed up differences between the representation that the translator expected and the one that the parser provided. The investigation of these differences revealed problems in the design of the AST framework that motivated a migration to the CZT framework.

Regression testing

JUnit was used to perform repeated tests after changes in the system, such as the migration to the CZT framework, bugfixes in the parser, and small redesigns. The execution of regression testing was important to make sure that those changes would not have an impact on the test cases for which the tool was already working correctly. JUnit provides a test runner with a graphical interface that helps to visualize which test cases were successful and which were not.

Acceptance testing

Members of the *Circus* group were asked to test *JCircus* and make comments on the usability and the usefulness of the tool. The suggestions motivated the implementation of a functionality in the tool that would generate a graphical interface for the translated programs, in order to provide a simple visualisation for the behaviour of those programs, and a way of interaction with the user.

4.5 Final considerations

In this section, we described *JCircus*, the tool we have implemented to automatise the translation strategy. We have provided a documentation of the development phases: requirement analysis, design and implementation and tests. We have also demonstrate how *JCircus* can be used.

JCircus uses the CZT framework. This choice was motivated by the quality of the project. The CZT framework is being used in other *Circus* projects. One is the type checker that *JCircus* already uses. The other is a model checker for *Circus*. The fact that the tools being implemented in the *Circus* group use the same framework is a great benefit, as will allow future integration.

Future work on *JCircus* includes the possibility of integration with the theorem proving module of the model checker. One of the extensions we plan for *JCircus* is the support for more types, beyond free types and the number type $\mathbb{A}$ that are currently supported. This requires support for more elaborated types of construct, for instance, set comprehension. The theorem prover would be able to determine the elements of a set comprehension; this information would be used to provide a Java representation for a set comprehension within the program generated by *JCircus*.

Members of the *Circus* team have contributed to extend the CZT java-core library for *Circus*. We have not extended the CZT parser yet, but this is in our plans; currently we are using another parser, but it constructs an AST tree that uses the CZT AST classes.

JCircus is available at [16]. The weppage also The binary distribution provides a short documentation on how to use the tool. A more complete documentation for users, with details on the translation laws, requirements on the input specification discussed in this thesis and the LaTeX format for the input file, is planned.

Chapter 5

Evaluation of the original translation strategy

The implementation of *JCircus* raised several questions related to the translation strategy and its formalisation. In this chapter, we explain the problems we found in the original strategy and how we solved them. Our work increased our confidence that the translation strategy preserves the correctness of *Circus* programs. We also consider alternatives to the implementation of some rules.

5.1 Errors in the original strategy

During the implementation of our tool, we found problems in the translation strategy related to the treatment of types, the parallelism of actions, and compound processes. In what follows, we discuss each of the issues raised.

5.1.1 Translation of *Circus* types

Types in Z are special kinds of sets. We can define types using basic types and a restricted set of constructors to build compound types. The ISO Standard Z defines the only basic type that the language pre-defines: the given type $\mathbb{A}$, which provides values for specifying number systems; the name $\mathbb{N}$, for example, is defined as having type $\mathbb{P}\,\mathbb{A}$. All other basic types are defined by the user or introduced with the mathematical toolkit, which is a library of mathematical definitions that the ISO Standard Z provides. Basic types can be composed to form complex types using the operators for powerset, cartesian product and schema construction. *Circus* adopts the type system of Z. Therefore, we can refer to *Circus* types and Z types indistinguishably.

The original translation strategy [29] defined an abstract class `Type` which is a superclass for all types defined within the system. It translates free type definitions and special forms of abbreviations as Java classes that represent types (that is, extend from `Type`).

This is the first difference between the original strategy and our implementation: the notion of "type". Abbreviations do not define types in Z; they define expressions. They are, as the name suggests, a simplified way of referring to an expression. Of course, they can be used to define sets, which are expressions; the original translation strategy regards some special kinds of sets, defined using abbreviation, as types.

Not all sets define types in Z, however. In spite of the fact that we can use any expression that represent a set in the declaration of a variable, the actual type of that variable is defined by the notion of maximal type. Therefore, if we define $a : \{1, 2, 3\}$ and $b : \{4, 5\}$, both variables have the same type $\mathbb{A}$. In *Circus*, we could, for example, assign a to b and vice-versa; this could lead to other types of inconsistencies, but would be correct with respect to typing. Mapping the same *Circus* type into different Java classes could result in code that, despite being correctly typed, would not compile. Because of this, we decided to take the approach of having a 1-1 mapping between *Circus* types and the Java classes that represent them.

Moreover, the original strategy presents another problem regarding the architecture that results from the translation of such sets that represent types. One requirement of the original strategy is that abbreviations could be defined in terms of at most one other set, and this set had to be a free type or another abbreviation. They could have the form $TName_{exp} == TName \cup \{ V_{n+1}, \ldots, V_m \}$ or $TName_{exp} == TName \setminus \{ V_{n+1}, \ldots, V_m \}$. This requirement implies that all abbreviations are extensions or restrictions of some free type. The free types would introduce a set of elements; and the abbreviations would be defined in terms of a free type or another abbreviation, by expanding or restricting its set.

The architecture for Java classes that represented types was based on the notion of extension/restriction of such sets. If a set A extended a set B (that is, all elements of B were in A), this would be translated as a Java class A that is a superclass of a class B (B extends A). For this reason, two different sets could not extend the same set, because this would lead to multiple inheritance, which is not allowed in Java.

To translate a free type definition or an abbreviation, we needed to check if it was extended by a previously defined set or not. If this is not the case, it would extend the class `Type` and we would declare a static constant for each of the elements in the free type or abbreviation, and constants `MIN_<Type>` and `MAX_<Type>` for the minimum and maximal values of the type; otherwise, it would extend the class of its expanding set, and redefine the constants for maximum and minimum values.

The original translation strategy [29] takes this specification as an example:

$$T_1 ::= A \mid B \mid C$$
$$T_2 == T_1 \cup \{D, E\}$$

The translation of these two types creates two classes: the class `T_2`, which represents the type T_2, extends the class `Type`; and the class `T_1`, which represents the type T_1, extends the class `T_2`. The class diagram can be seen in Figure 5.1(a).

If, instead, we had T_2 restricting T_1, as in the following specification:

$$T_1 ::= A \mid B \mid C$$
$$T_2 == T_1 \setminus \{C\}$$

then the inheritance relation would be inverted, as we see in Figure 5.1(b).

There is a problem in the first example, however: it will not pass in the typechecker. The names D and E have not been declared; but even if they had been declared, they would not have the same type as the elements of T_1. Therefore, the expression $T_1 \cup \{D, E\}$ is wrongly typed.

We could make that specification correct if we included D and E in the definition of the free type T_1:

$$T_1 ::= A \mid B \mid C \mid D \mid E$$
$$T_2 == T_1 \cup \{D, E\}$$

Now, it is correctly typed. But what is the point in defining a new set T_1 like this? It represents exactly the same set as T_1. We could use the union set operator to define a set in terms of another one that has been restricted, as in the following example:

$$T_1 ::= A \mid B \mid C \mid D \mid E$$
$$T_2 == T_1 \setminus \{C, D, E\}$$
$$T_3 == T_2 \cup \{D, E\}$$

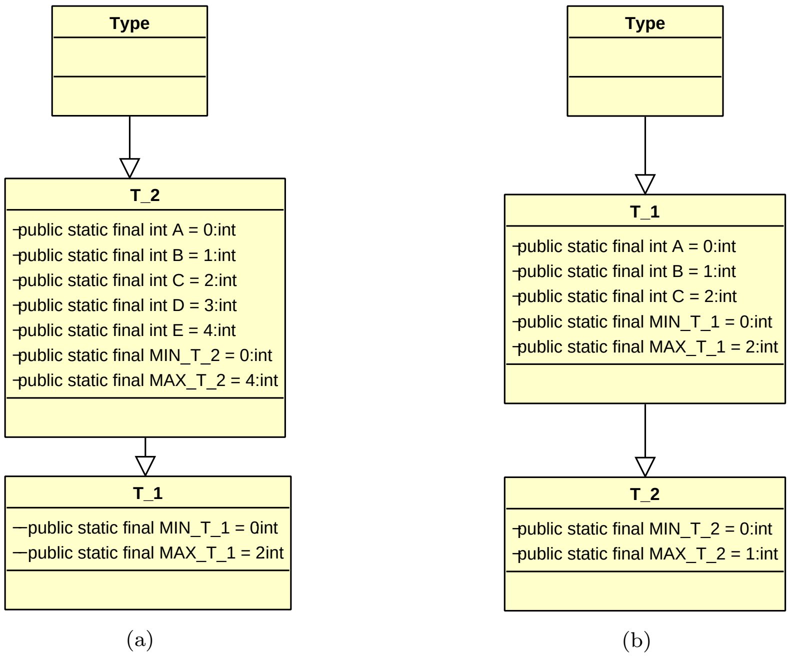

Type
T_2
-public static final int A = 0:int
-public static final int B = 1:int
-public static final int C = 2:int
-public static final int D = 3:int
-public static final int E = 4:int
-public static final MIN_T_2 = 0:int
-public static final MAX_T_2 = 4:int
T_1
—public static final MIN_T_1 = 0int
—public static final MAX_T_1 = 2int
(a)
Type
T_1
-public static final int A = 0:int
-public static final int B = 1:int
-public static final int C = 2:int
-public static final MIN_T_1 = 0:int
-public static final MAX_T_1 = 2:int
T_2
-public static final MIN_T_2 = 0:int
-public static final MAX_T_2 = 1:int
(b)

Figure 5.1: Architecture for types

This would lead to a situation where the class for T_2 would have to extend both from T_1 and T_3 (because, T_2 restricts T_1 and T_3 expands T_2; see Figure 5.2). But one of the requirements was that different sets should not extend the same set. We see then that defining a set with an abbreviation using the set union operator in a way that is permitted by the original translation strategy actually invalidates the translation because leads to a situation that is not permitted.

In our strategy we adopted a simplified way to represent types. Types are only introduced with free types, and they are translated into classes that extend the abstract class `Type`. The other possible type is `CircusNumber` which represents the number sets $\mathbb{N}$ and $\mathbb{Z}$, and also extends `Type`.

We understand, however, that it would be interesting to differentiate, in some way, the sets used to define the types of variables. Consider the following example:

channel $c : \mathbb{N}$

process $P_1 \mathrel{\widehat=}$ **begin**
    $\bullet\ c!(-1) \rightarrow Skip$
end

process $P_2 \mathrel{\widehat=}$ **begin**
    $\bullet\ c?x \rightarrow Skip$
end

process $P_3 \mathrel{\widehat=}\ P_1 \llbracket c \rrbracket P_2$

Here we declare a channel c that communicates natural numbers. Process P_1 writes a negative value on c, and process P_2 reads a value from channel c. Process P_3 is a parallel composition of

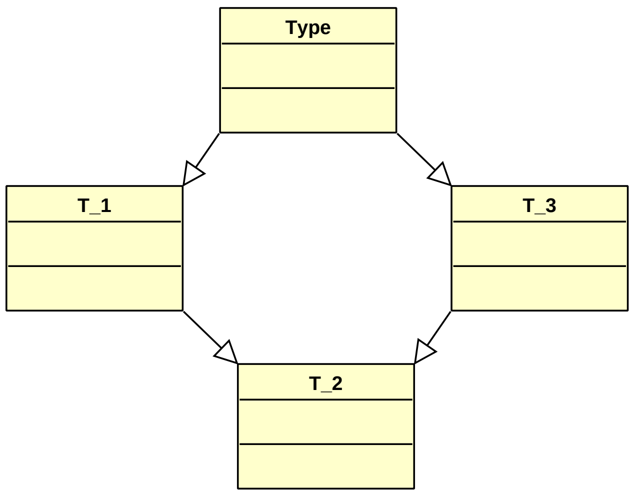


Figure 5.2: Architecture for types

P_1 and P_2, in which the value would be transmitted from P_1 and P_2. This example is correctly typed, but the execution of P_3 will result in deadlock, because we are trying to transmit a value that is not in the set of values that c can communicate.

Implementing this is not an easy task, because the set used to declare c could be defined in terms of, for example, a set comprehension as $\{x : \mathbb{N} \mid x > 5 \land x < 10\}$, and in fact, the predicate could be arbitrarily complex, which would require a theorem prover to determine the elements of the set. We could restrict the forms of expressions that define sets in a way that would make it easy to determine the elements of the set, and record this information together with the type information, for each variable and channel. In this case, we could implement the correct behaviour for some simple cases, like the example mentioned. This is an interesting piece of future work.

5.1.2 Action parallelism

The translation of action parallelism, recursion, and iterated actions requires an environment for local variables, *LocalEnv*, which maps a variable name to its type. The local variables for an action are all the variables that are in scope when it is defined, including process parameters, action parameters, variables introduced in a variable block, but not state components.

Action parallelism/interleaving is a bit different from process parallelism/interleaving. The former requires the definition of the set of variables which each parallel action can modify. Parallel actions cannot modify the same variable, which means that these sets must be disjoint; we will call them partitions. Besides that, parallel actions deal with copies of the local variables, so that one action that writes on a variable does not interfere on the other action if it reads the same variable. At the end of the parallelism, the variables are updated with the final values of their respective copies from the actions where they appear in the partition.

Consider the following example:

$$\begin{array}{l}
\textbf{process } P \mathrel{\widehat=} \textbf{ begin} \\
\quad \textbf{state } S \mathrel{\widehat=} [x, y, z : \mathbb{N}] \\
\quad \mathit{Init} \mathrel{\widehat=} x, y, z := 0, 0, 0 \\
\quad A \mathrel{\widehat=} (y := 2) \mathbin{|\![} y \mid x \mathbin{]\!|} (x := y + 5;\ z := 10) \\
\quad \bullet\ \mathit{Init};\ A \\
\textbf{end}
\end{array}$$

This is a basic process, with a state schema that defines state components x, y and z, an initialisation action *Init*, and an action A defined using an interleaving. The main action performs the initialisation first, and then the interleaving. The local environment for the interleaving is composed by x, y and z. At the moment the interleaving is executed they all have value zero, and their values are assigned to copies of the variables for each of the parallel actions. Therefore, at the end of parallelism, x will always have value 5, even if the assignment to y occurs first, because it is dealing with the copy of y, which is different from the one that the left action is updating.

At the end of the interleaving, y will be assigned the value of its copy from the left action, therefore, 2; x will be assigned the value of the copy from the right action, therefore, 5; and z will be assigned no value at all, because it is not in any of the sets. So, the assignment $z := 10$ is useless and does not produce any effect; z has value zero at the end.

The implementation of parallel actions reflects this semantics. The translation of action A in our example is shown in Figure 5.3. It declares and instantiates two inner classes that implement `CSProcess`, one for the left action (lines 2-10), and the other for the right action (lines 11-21). These classes declare and initialise attributes which represent the copies of the local variables (lines 3-5 and 13-15). The method `run` of each class contains the implementations of the actions (lines 7-9 and 17-20), which only deal with the copies of the variables. The objects are given as parameters to a `Parallel` object, which is run (line 26). After the parallelism terminates to run, the values of the copies are retrieved from the respective sets (lines 27 and 28).

The rule used in the original translation strategy for handling parallel or interleaved actions is presented below; it contains some mistakes.

Rule *Action parallelism/interleaving (original)*

$[\![A_1 \mathbin{[\![} ns_1 \mid cs \mid ns_2 \mathbin{]\!]} A_2]\!]^{Action} =$

```
class LName implements CSProcess {
    (IAuxVars (ns₁ \ (dom λ)) ind L) (DeclLcVars λ ind L)
    public LName((LcVarsArg λ)) { ILcVars λ ind L }
    public void run() { RenVars [[ A₁ ]]^Action (ns₁ ∪ (dom 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*

Only state components that take part in the partition set are being declared in the inner class and renamed in the translation of the action. The other state components are being directly accessed in the class, which means that if the action is writing on these variables, they are not writing on a copy, and therefore, are permanently assigning on the variable. In the example above, this means that variable z would hold value 10 after the parallelism terminates to run. The solution

```
private void A(){
    class ParallelLeftBranch_0 implements CSProcess{
        public CircusNumber aux_left_y_0 = y;
        public CircusNumber aux_left_z_0 = z;
        public CircusNumber aux_left_x_0 = x;
        public ParallelLeftBranch_0 (){}
        public void run(){
            aux_left_y_0 = new CircusNumber(2);
        }
    }
    CSProcess left_0 = new ParallelLeftBranch_0 ();
    class ParallelRightBranch_0 implements CSProcess{
        public CircusNumber aux_right_y_0 = y;
        public CircusNumber aux_right_z_0 = z;
        public CircusNumber aux_right_x_0 = x;
        public ParallelRightBranch_0 (){}
        public void run(){
            aux_right_x_0 = aux_right_y_0.add(new CircusNumber(5));
            aux_right_z_0 = new CircusNumber(10);
        }
    }
    CSProcess right_0 = new ParallelRightBranch_0 ();
    CSProcess[] processes_0 = new CSProcess[]{
        left_0, right_0
    };
    (new Parallel(processes_0)).run();
    y = ((ParallelLeftBranch_0)processes_0[0]).aux_left_y_0;
    x = ((ParallelRightBranch_0)processes_0[1]).aux_right_x_0;
}
```

Figure 5.3: Implementation of action A

is to make copies of all local variables and also state components, and also rename them in the translation of the body of the method run.

The other problem is that just renaming the variables in the code that results from the translation of A_1 and A_2 is not enough, because these actions can contain a call to another action. In this case, the action which is called does not deal with the copies of the variables, but with the original ones, because it does not receive the copies as parameter.

The solution requires that the action calls made inside an action parallelism are translated into an instantiation and call to the method `run` of an inner class. The inner class defines as attributes the state components and process parameters of the basic process in which the parallelism is defined, which are defined in a new environment *BasicProcEnv*. The method `run` of the inner class contains the translation of the action, and it deals with the copies of the variables, and not the original ones.

The corrected rule for parallelism (Rule A.41) and interleaving (Rule A.42) are presented in the Appendix A.

5.1.3 Compound processes

The translation strategy translates a basic process (that is, a process defined between the keywords **begin** and **end**) as a call to the method run of an anonymous implementation of `CSProcess` (see Rule A.6). This translation is very appropriate because a basic process, as a Java class, is an entity that encapsulates data and behaviour. The correspondence is perfect:

- state components are translated as class attributes (data);
- action definitions are translated as private methods (behaviour). The methods are private because an action cannot be called from outside the basic process where it is defined;
- the main action is translated as the body of the method `run` (main behaviour). In the JCSP, a process is an instance of a class implementing the `CSProcess` interface and its behaviour is defined by the `run` method.

A basic process is an element of the syntactic category Process as defined in the *Circus* grammar (see Figure 2.2). In order to have a consistent strategy, all constructs of the same *Circus* syntactic category must be translated to a construct of the same Java syntactic category: process declarations (ProcDecl) are translated as Java classes, in which the process name N is the name of the class; process definitions (ProcDef) are translated as the code for attribute, constructor and method `run` definitions for the process declaration in which they occur; action definitions (ActionDef) are translated as private methods, as explained before; and actions (Action) are translated as sequential Java code.

So, in order to be consistent with the translation of basic process discussed before, all constructors in the *Circus* syntactic category Process must be translated as a call to the method `run` of an instance of a class implementing the `CSProcess` interface. The original translation strategy did not reflect this in some rules for compound processes, namely, the rules for sequential composition of processes, implicit parametrised process invocation and internal choice of processes. These were translated as sequential Java code, as we will see.

The rule for implicit parametrised process invocation declares an inner class (*DeclareProcessClass*) and instantiates it, passing as parameters to the constructor the actual parameters and the variables in the local environment.

Rule *Implicit parametrised process call (original)*

```
$|[(Decl \bullet Proc)(e_1, \ldots, e_n)]|^{Proc} =$
    $DeclareProcessClass\ Decl\ Proc\ index$
    I_$index$ i_$index$_$index$ =
        new I_$index$(($JExp\ e_1$),...,($JExp\ e_n$),
                      ($JList\ (ListFirst\ LocalVarEnv)$));
    i_$index$_$index$.run();
```

The rule for sequential composition of process is simply translated as Java sequential composition code. In a sequential composition of processes, each process has its own underlying state.

Rule *Sequential composition (original)*

$$|[\,Proc_1; \ldots ;Proc_n\,]|^{Proc} = |[Proc_1\,]|^{Proc}\ ;\ \ldots\ |[\,Proc_n]|^{Proc}$$

The rule for internal choice uses class `RandomGenerator` to create a pseudo-random number, which will be checked in a switch statement to decide which action will be executed.

Rule *Internal choice (original)*

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

The problems with these rules were revealed when they were applied in combination with the rule for process parallelism. The translation of a parallelism uses the class `Parallel` and instantiates an array containing the parallel processes.

Rule *Process parallelism (original)*

```
$|[Proc_1 \,|[\, CSExp \,]|\, Proc_2]|^{Proc} =$
    (new CSProcess(){
        public void run() {
            new Parallel (
                    new CSProcess[] {$|[\,Proc_1\,]|^{Proc}$ , $|[\,Proc_2\,]|^{Proc}$}
                ).run ();
        }
    }).run();
```

The function $|[\_]|^{Proc}$ is being called inside an instantiation of an array of elements of type `CSProcess`. Therefore, what we expect to receive as the result of this function is an expression for an instance of a `CSProcess`, and not sequential Java code. The translation of this example

process $P \mathrel{\widehat=} (A;\ B)\ |[\, ch \,]|\ C$

would result in a Java class called P with the following code as the body of its method `run`:

```
(new CSProcess() {                                          (1)
    public void run() {                                     (2)
        new Parallel(                                       (3)
            new CSProcess[] {                               (4)
                (new A( ... )).run(); (new B( ... )).run();,    (5)
                (new C( ... )).run();                       (6)
            }                                               (7)
        ).run();                                            (8)
    }                                                       (9)
}).run();                                                  (10)
```

This code does not compile because sequential Java code appears where an instance of `CSProcess` is expected (lines 5 and 6).

Part of the solution was to encapsulate the sequential code into an anonymous instantiation of a `CSProcess`. The rule for sequential composition, for example, becomes:

Rule A.16 *Sequential composition*

```
$[\![ Proc_1;\ Proc_n ]\!]^{Proc} =$
    (new CSProcess(){
        public void run() {
            $[\![Proc_1]\!]^{Proc}$ ; $[\![ Proc_n ]\!]^{Proc}$ }
    }).run();
```

The other rules are modified in a similar way, see Appendix A. Now, all constructs of the *Circus* syntactic category Proc are translated to the same sort of Java code.

However, there is also a problem in the rule for parallelism. As mentioned above, the call to the function $[\![\_]\!]^{Proc}$ in the rule for parallelism should return Java code for an instantiation of a *CSProcess*, and not the call to the method `run` of an instance of *CSProcess*. Therefore, we should enclose the call to the function $[\![\_]\!]^{Proc}$ in an instantiation of an inner class.

Rule *Process parallelism (new version)*

```
$[\![Proc_1 \ [\![\ CSExp\ ]\!] \ Proc_2]\!]^{Proc} =$
    (new CSProcess(){
        public void run() {
            new Parallel (
                    new CSProcess[] {
                        new CSProcess() { public void run() { $[\![ Proc_1 ]\!]^{Proc}$ } },
                        new CSProcess() { public void run() { $[\![ Proc_2 ]\!]^{Proc}$ } }
                    }
            ).run();
        }
    }).run();
```

The rules for sequential composition, internal choice and implicit parametrised invocation were wrong because the code they returned was not encapsulated inside a `(new CSProcess() { ... } ).run();`. The rules for process invocation and parametrised process invocation were not wrong, but they defined an unnecessary instantiation of a `CSProcess`.

Rule *Process call (original)*

```
$[\![ N ]\!]^{Proc}$ =
    (new CSProcess(){
        public void run() {
            (new N($ExtractChans\ VisChanEnv$)).run();
        }
    }).run();
```

Rule *Parametrised call (original)*

```
$[\![ N(e_1, \ldots, e_n) ]\!]^{Proc}$ =
    (new CSProcess(){
        public void run() {
            (new N(($JExp\ e_1$), ... ,($JExp\ e_n$),
                    ($ExtractChans\ VisChanEnv$))).run();
        }
    }).run();
```

In the translations above, an anonymous instantiation of a `CSProcess` defines, in the method `run`, an instantiation of an object of class `N`, which is itself a `CSProcess`. Therefore, the anonymous call is unnecessary; it does nothing than just creating one more thread in the system. In the rule for parallelism, we can find the same situation, because class `Parallel` is also a subclass of `CSProcess`. The rules above, and the one for parallelism can then be simplified to avoid the anonymous call.

Rule A.7 *Process call*

```
$[\![ N ]\!]^{Proc}$ = (new N($ExtractChans\ VisChanEnv$)).run();
```

Rule A.8 *Parametrised call*

```
$[\![ N(e_1, \ldots, e_n) ]\!]^{Proc}$ = (new N(($JExp\ e_1$), ... ,($JExp\ e_n$),
                                   ($ExtractChans\ VisChanEnv$))).run();
```

Rule A.18 *Process parallelism*

```
$[\![ Proc_1 \mathbin{[\![} CSExp \mathbin{]\!]} Proc_2 ]\!]^{Proc}$ =
    new Parallel (
            new CSProcess[] {
                new CSProcess() { public void run() { $[\![ Proc_1 ]\!]^{Proc}$ } },
                new CSProcess() { public void run() { $[\![ Proc_2 ]\!]^{Proc}$ } }
            }
        ).run ();
```

As mentioned in Section 4.1.2, a parallelism does not take into account the channels defined in the operator; rather, the parallel processes or action synchronise of all channels that they have in common. Therefore, the synchronisation set *CSExp* is not taken into account in the Rule A.18.

The modification in the above rules provide both simplification of the code generated and also an optimisation, because there is no creation of an unnecessary process.

5.2 Alternatives for implementation

In this section we discuss rules that we have implemented differently from the original proposal. Some rules for multi-synchronisation were modified to cover more general cases.

5.2.1 Multi-synchronisation

In the original translation strategy, the code generated for a program involving multi-synchronisation is completely different from the code generated if only simple synchronisation were used. For instance, multi-synchronisation uses channels *to* and array *from* to carry out the communication, whereas simple synchronisation requires only one `Any2OneChannel`; the `read` and `write` methods used in the case of simple synchronisation were substituted by instantiation of clients.

The code for a basic process, however, is contained inside a class, which is a `CSProcess` that implements the basic process. At the time we are translating a basic process we do not know whether the channels that it uses take part in a multi-synchronisation or not. Consider the following example:

$$\textbf{channel}\ ch : \mathbb{N}$$

$$\textbf{process}\ A \mathrel{\widehat=} \textbf{begin}\ \bullet\ ch?x \rightarrow \mathit{Skip}\ \textbf{end}$$

The translation of process A is a Java class with the same name. This class has a channel ch as attribute, and its method `run` contains an input communication on ch.

If channel ch took part in a multi-synchronisation, however, the declaration of channel ch should have been replaced by `private Any2OneChannel[] from_ch` and `private Any2OneChannel to_ch`, and the code for communication should also have been changed accordingly.

However, to determine whether ch is used in a multi-synchronisation or not requires inspecting the uses of A. In process $ParABC$ defined below, ch is multi-synchronised, whereas in process $ParAB$, it is not.

$$\textbf{process}\ B \mathrel{\widehat=} \textbf{begin}\ \bullet\ ch!1 \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ C \mathrel{\widehat=} \textbf{begin}\ \bullet\ ch?x \rightarrow \mathit{Skip}\ \textbf{end}$$

$$\textbf{process}\ \mathit{ParAB} \mathrel{\widehat=} A \mathbin{[\![} \{\!|\ ch\ |\!\} \mathbin{]\!]} B$$

$$\textbf{process}\ \mathit{ParABC} \mathrel{\widehat=} A \mathbin{[\![} \{\!|\ ch\ |\!\} \mathbin{]\!]} B \mathbin{[\![} \{\!|\ ch\ |\!\} \mathbin{]\!]} C$$

Therefore, we can see that it is the processes that use A that have the information necessary to know if the channels that A uses are multi-synchronised or not.

In our implementation, we define a class `GeneralChannel` (introduced in Chapter 3), which is used instead of `Any2OneChannel`. It encapsulates the data and algorithms for the multi-synchronised and simple-synchronised cases. The translation rules for channel communications just need to call the methods `read` or `write` from `GeneralChannel`. These methods execute the code for simple or multi-synchronised case, according to the values of the attribute that are set up in the constructor. They are set up by the compound process that instantiates its sub-processes, and have the knowledge of how their channels should be used.

5.2.2 Synchronisation channels

A similar problem occurs when we have to define if a synchronisation channel will be used as a reader or a writer by a particular process. The following example shows three processes A, B and

C that uses a channel ch and are combined in parallel two by two in the compound processes $ParAB$, $ParAC$ and $ParBC$.

$$\mathbf{process}\ A \mathrel{\widehat=} \mathbf{begin}\ \bullet\ ch \rightarrow Skip\ \mathbf{end}$$

$$\mathbf{process}\ B \mathrel{\widehat=} \mathbf{begin}\ \bullet\ ch \rightarrow Skip\ \mathbf{end}$$

$$\mathbf{process}\ C \mathrel{\widehat=} \mathbf{begin}\ \bullet\ ch \rightarrow Skip\ \mathbf{end}$$

$$\mathbf{process}\ ParAB \mathrel{\widehat=} A \llbracket \{\!|\ ch\ |\!\} \rrbracket B$$

$$\mathbf{process}\ ParAC \mathrel{\widehat=} A \llbracket \{\!|\ ch\ |\!\} \rrbracket C$$

$$\mathbf{process}\ ParBC \mathrel{\widehat=} B \llbracket \{\!|\ ch\ |\!\} \rrbracket C$$

Channel c is not used for input or output. How to determine which process calls method `read()` on `ch` and which calls `write(null)`, in each parallelism? The original translation strategy defined an environment *ChanUseEnv* (available for each process) that mapped a channel name to its use (*Input*, *Output* or *AltInput*). If a synchronisation channel were classified as *Input* for a particular process, the process should call `read()`; otherwise, it should call `write(null)`. However, the strategy did not make clear how this classification should be made; it just assumed that the classification was provided.

In this example, we see that this classification should not be statically determined, because in this case, one of the parallelisms would contain two readers, or two writers. Our solution was to define another type, *Undefined*, to classify synchronisation channels. The class `GeneralChannel` also defines the method `synchronise`, which is used by synchronisation channels instead of `read` and `write`. The method `synchronise` calls `read` or `write`, according to the information in `ChanInfo`, which determines if this instance should be used as a reader or as a writer. The decision of whether the process will be the reader or the writer in a parallelism is left to the instantiation, when `ChanInfo` is initialised.

5.2.3 Recursion and iterated operations for actions

The translation of recursive actions and iterated actions uses inner classes. As with action parallelism, it is also necessary to use the information in the local environment make copies of the local variables. The motivation to have copies of local variables here is not the same as in the translation of the parallelism; it is just because some kinds of variables cannot be seen inside the inner class.

The original translation strategy determines that, for each local variable, there should be an attribute in the inner class which would be initialised with the original value, in the constructor, and then retrieved after the action terminates. However, state components and process parameters can be seen inside the inner class because they are translated as attributes of classes. In our implementation, instead of declaring copies of these variables we just access them directly in the inner class. The new rules that we actually use are presented in Appendix A (Rules A.43, A.49 and A.50).

5.2.4 External choice

The original strategy presented an alternative implementation for the external choice operation, when the actions were guarded and the guards were mutually exclusive. This can be refined to an if-then-else statement, and, consequently, has a simple implementation.

We chose not to implement this rule because it is not trivial. The problem of determining if predicates are mutually exclusive is a task for a theorem prover. The rule is still implemented correctly with the standard rule for external choice, because only one guard will be enabled, and therefore, only one action could be selected. If the user wants a more efficient implementation, they can use a simple refinement rule to transform the external choice into *Circus* guarded actions. This is in accordance with the spirit of the translation strategy, which should be applied to a refined *Circus* program.

5.3 Final considerations

In this chapter we have discussed some errors found in the original translation strategy, and the solutions that we have proposed. We have also considered alternative implementations for some rules.

One interesting point that arose from our work is the difference between the implementation of types in the original translation strategy, and in our new proposal. In the original strategy, types were regarded as sets, and variables defined using different sets yielded variables from different Java types. Our approach is closer to the concept of type in *Circus*: only the maximal types are regarded as types. For now, only free types and numbers are considered, but we intend to keep with this philosophy when we extend the strategy to deal with complex types.

We have also found out that the implementation of action parallelism did not correspond to the semantics of *Circus*. Action parallelism deals with copies of the values. After the parallelism terminates, the original values are updated with the copies from each parallel action, according to their partition sets. When a parallel action contains a call to another action, this must be implemented with an inner class whose attributes are the copies of the variables. This is a more complex and less intuitive implementation, than the original proposal, which uses a call to a method; however, it was necessary in order to have a correct implementation.

The main modification to the original translation strategy was the use of `GeneralChannel` to represent a channel, instead of `Any2OneChannel`. It permitted the unification of rules for simple and multi-synchronisation cases, and an easy way to determine if a channel that is not defined as input nor output is to be used as a reader or a writer in each process where it takes part.

Chapter 6

Verification of the multi-synchronisation protocol

In the last chapter, we revealed some errors that were found in the original translation strategy. A complete proof of soundness for the translation strategy requires a formal semantics for Java, and a mapping from the *Circus* semantics. With that, we could prove that the semantics of every *Circus* program is in correspondence with the semantics of the Java program obtained with the translation. This, however, is by no means a simple task. Here, we propose a smaller step to bridge the gap between *Circus* and Java: to model the JCSP constructs and the Java programs in *Circus* itself, and use the *Circus* refinement calculus to prove that the translation rules are refinement laws. We illustrate this approach by considering the algorithm for multi-synchronisation.

We present a *Circus* model of multi-synchronisation in a channel that takes part in an external choice. Afterwards, we present a *Circus* model for the implementation of the multi-synchronisation protocol, which is based on the Java source code generated by our translation strategy. We then prove, using the refinement calculus of *Circus* and the strategy presented in [42], that the multi-synchronisation is refined by our model of the implementation.

The translation strategy and *JCircus* can handle more general forms of multi-synchronisation than that we consider here. Clients may take part in more than one multi-synchronisation, in more than one simple synchronisation, and values may be carried through multi-synchronised channels. However, the actual rule for translation of multi-synchronisation is just a generalisation of the one we verify here. We believe that its verification could be carried out using a similar approach.

6.1 Multi-synchronisation in *Circus*

We model a multi-synchronisation as an iterated parallelism of n processes that synchronise on a channel m. The processes are indexed by elements of the set I, which range from zero to $n - 1$. The constants n and I are introduced in axiomatic definitions.

$$\begin{array}{|l} n : \mathbb{N} \\ I == 0 \mathrel{..} (n-1) \end{array}$$

The multi-synchronised channel m is declared as an untyped channel, as it carries no value.

channel m

The model also uses channels that represent individual events. These are declared as having type I, because they carry the index of the process that is performing the event.

$$\mathbf{channel}\ q, \mathit{interrupt} : I$$

The multi-synchronisation is defined in the process *Network_Spec*, which follows.

$$\mathbf{process}\ \mathit{Network\_Spec} \mathrel{\widehat=} \left(\begin{array}{l} \| \ i : I \bullet \lbrack\!\lbrack \{m\} \rbrack\!\rbrack \\ \quad (\mu X \bullet \\ \qquad m \rightarrow\ q.i \rightarrow\ X \\ \qquad \Box \\ \qquad \mathit{interrupt}.i \rightarrow\ X \\ \quad) \end{array} \right) \setminus \{\!|\ m\ |\!\}$$

The network is the parallel combination of a collection of processes, indexed by variable i, which range from zero to $n-1$. The i-th process is recursive, and continuously offers the choice between the multi-synchronised event m, and the individual event $\mathit{interrupt}.i$. The multi-synchronisation is followed by the event $q.i$, which represents an individual transaction that every process executes after the multi-synchronisation. The multi-synchronised channel m is hidden from the environment, which means that we know the identities of all of its participants: no other process outside the network will ever synchronise on it.

The external behaviour of this network can be informally described as follows: the events $\mathit{interrupt}.i$ (for all i in I) will be continuously performed, in any order, allowing repetition; but once an event $q.i$ occurs (for any i in I), the events $q.i$ for all other processes have to be executed only once before any event $\mathit{interrupt}.i$ is executed again.

6.2 A *Circus* model for the multi-synchronisation protocol

As already mentioned, the model for the multi-synchronisation protocol is based on the Java source code generated by the translation. The code was written using JCSP; the classes `MultiSyncClient` and `MultiSyncControl` are implementations of the interface `CSProcess` and their methods `run` make use of the JCSP classes for parallelism and alternative. There is a quite direct correspondence between our model and the actual implementation.

Figure 6.1 shows the *Circus* model for the protocol. The architecture was explained in Chapter 3, and the idea is the same for our simplified version. The implementation is a parallelism between the controller (the second process), which is implemented by an instance of `MultiSyncControl`, and an interleaving of clients (the first process), which are implemented by instances of `MultiSyncClient`.

The controller is the process that manages the requests for multi-synchronisation made by the clients. Since in our simplified version we deal with only one multi-synchronised channel, our model contains only one controller. It communicates with the clients through channels to_A, from_A, to_B and from_B; these communications represents the four phases of the protocol. In our model, we deal with four channels, instead of reusing the implementation's channels *to* and *from*, in the first and third, and second and fourth phases of the protocol, respectively.

The channels *from* are arrays that each controller uses to communicate with its clients. The channels *to* are used by the clients to communicate values to the controller; it is a shared channel, however, it is not multi-synchronised, because the clients synchronise with the controller one at a time.

Channels from_A and from_B carry values from the set $I \times \mathit{Boolean}$. The type *Boolean* is introduced as a free type. Remember that channel *from*, in the implementation, is an array of

```
$\textbf{process}\ Network\_Impl \mathrel{\widehat=}$
  $\Bigg( \Bigg( \interleave\ i : I \bullet$
        $\mu X \bullet to_A!i \rightarrow$
              $from_A.i?any \rightarrow$
                  $to_B!i \rightarrow$
                      $from_B.i?synchronised \rightarrow$
                          $(synchronised = true)\ \&\ q.i \rightarrow\ X$
                          $\Box$
                          $(synchronised = false)\ \&\ X$
              $\Box$
              $interrupt.i \rightarrow$
                  $to_A!(flip\ i) \rightarrow X$
                  $\Box$
                  $from_A.i?anyt \rightarrow to_B!(flip\ i) \rightarrow from_B.i?any \rightarrow X \Bigg)$
     $[\![ \{ to_A, from_A, to_B, from_B \} ]\!]$
    $\Bigg( (\mu X \bullet count : I \bullet$
        $(count > 0 \land count \leq n)\ \&$
            $to_A?nextOffer \rightarrow$
                $(nextOffer \geq 0)\ \&\ X(count - 1)$
                $\Box$
                $(nextOffer < 0)\ \&\ X(count + 1)$
        $\Box$
        $(count = 0)\ \&$
            $(\mu Y \bullet i : I \bullet$
                $(i < n)\ \&\ from_A.i!true \rightarrow\ Y(i + 1)$
                $\Box$
                $(i = n)\ \&$
                    $(\mu Z \bullet i, count : I \bullet$
                        $(i \geq 0 \land i < n)\ \&\ to_B?nextcommit \rightarrow$
                            $(nextcommit \geq 0)\ \&\ Z(i + 1, count - 1)$
                            $\Box$
                            $(nextcommit < 0)\ \&\ Z(i + 1, count)$
                        $\Box$
                        $(i = n)\ \&$
                            $(\mu W \mathrel{\widehat=} i : I \bullet$
                                $(i < n)\ \&$
                                    $from_B.i!(count = 0) \rightarrow\ W(i + 1)$
                                $\Box$
                                $(i = n)\ \&\ X(n)$
                            $)(0)$
                    $)(0, n)$
            $)(0)$
      $)(n) \Bigg) \Bigg)$
  $\setminus \{\!|\ to_A, from_A, to_B, from_B\ |\!\}$
```

Figure 6.1: *Circus* model for the multi-synchronisation protocol

channels; that is why we defined its type as a cartesian product. The I represents the index of the array, and *Boolean* is the type of the values the channel carries.

$$Boolean ::= True \mid False$$

$$\textbf{channel } from_A, from_B : I \times Boolean$$

We also define an auxiliary function *flip*, which is introduced in an axiomatic definition.

$$\begin{array}{|l} flip : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall\, i : \mathbb{N} \bullet flip\ i = -(i+1) \end{array}$$

This function is used to invert the signal of the index of a process; channels to_A and to_B send a flipped index to signal to the controller that a client has given up participating in the multi-synchronisation. Their definitions are as follows.

$$Message == -n \,..\, (n-1)$$

$$\textbf{channel } to_A, to_B : Message$$

The multi-synchronisation protocol consists of four phases. In the first phase, clients use channel to_A to make offers to take part in the multi-synchronisation. The controller counts how many clients have not made an offer yet. It does not pass to the second phase until *count* is zero, that is, all clients have made an offer. The first phase is implemented by the first recursion, on X, in the controller.

In the second phase (recursion on Y), the controller invites the clients to commit to the multi-synchronisation. Clients have the chance to commit to the multi-synchronisation, by executing the event $from_A.i$, or to interrupt, by executing $interrupt.i$.

If all the clients decide to commit then the multi-synchronisation will occur. In the third phase (recursion on Z), the controller counts how many clients interrupted. In the fourth phase (recursion on W) the controller communicates to the clients if all clients agreed to synchronise, that is, if *count* is equal to zero. If this is the case, then the clients perform their $q.i$ events, to acknowledge that the multi-synchronisation has occurred and the protocol is reinitialised; otherwise, the protocol is just reinitialised.

In our model of the implementation, the channels used for communication between the controller and clients are hidden. This gives us an interface which is the same as the one for our model of the specification: only channels *interrupt* and q can be seen by the environment. The refinement consists in proving that every behaviour that can be observed in the specification model is also a behaviour that can be observed in the implementation model. This is what is guaranteed by the *Circus* refinement relation.

6.3 Proof of refinement

Here we present an overview of the steps used to prove that the specification is refined by the multi-synchronisation protocol. Each step is justified by the application of a refinement law; the complete refinement can be found in the extended version of this thesis [16]. We used the *Circus* laws already published in the literature [42, 32], but we also needed some new laws which we present in Appendix B. The proof of these new laws is left as future work.

The approach taken for carrying out the refinement consists in refining the specification to an action system; transforming the model for the implementation to another action system; and then proving that the action systems are equivalent. An action system is a recursive process in which only one event is performed in each iteration. The execution is controlled by a local variable, the program counter, whose value is used to check which event is enabled in each iteration. The transformation into an action system is a way to linearise the model. The strategy for refinement is summarised in Figure 6.2.

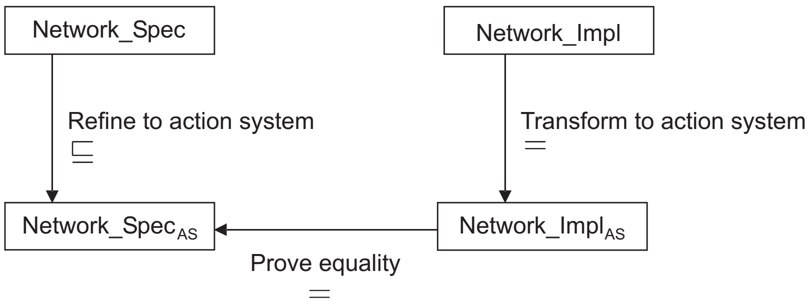


Figure 6.2: Strategy for proof of refinement

6.3.1 Refining *Network_Spec* to an action system

We start with our model for the specification of a multi-synchronisation.

$$\left(\begin{array}{l} \parallel i : I \bullet [\![\{m\}]\!] \\ \quad (\mu X \bullet \\ \qquad m \rightarrow\ q.i \rightarrow\ X \\ \qquad \Box \\ \qquad interrupt.i \rightarrow\ X \\ \quad) \end{array}\right) \setminus \{\!|\ m\ |\!\}$$

First, we refine each of the parallel processes to an action system. For that, we introduce a variable block that introduces a variable *pcs*, which is the program counter for each process *i*.

⊑ { Law B.1 (Action system conversion) }

$$\left(\begin{array}{l} \parallel i : I \bullet [\![\{m\}]\!] \\ \quad \textbf{var}\ pcs : \mathbb{N} \bullet \\ \qquad pcs := 0; \\ \qquad (\mu X \bullet \\ \qquad\quad pcs = 0\ \&\ m \rightarrow\ pcs := 1;\ X \\ \qquad\quad \Box \\ \qquad\quad pcs = 0\ \&\ interrupt.i \rightarrow\ pcs := 0;\ X \\ \qquad\quad \Box \\ \qquad\quad pcs = 1\ \&\ q.i \rightarrow\ pcs := 0;\ X \\ \qquad) \end{array}\right) \setminus \{\!|\ m\ |\!\}$$

Now we widen the scope of the counter *pcs*, by bringing its declaration to the outside of the parallelism. The variable *pcs* is now an array, indexed by the variable *i* from *I*.

= {Laws B.88 (Parallel state) and B.89 (Parallel assignment) }

$$\left(\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\ i : I \bullet pcs_i := 0); \\
\quad \|\ i : I \bullet [\![\{m\}]\!] \\
\qquad (\mu\, X \bullet \\
\qquad\quad pcs_i = 0\ \&\ m \rightarrow\ pcs_i := 1;\ X \\
\qquad\quad \Box \\
\qquad\quad pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \\
\qquad)
\end{array}\right) \setminus \{\!|\ m\ |\!\}$$

Now the parallel composition is transformed into a sequential action system. The event m, which was shared between the processes, gives rise to a prefixing whose guard is the conjunction of the processes' guards, and whose action is the interleaving of the processes' actions.

= { Law B.2 (Action system parallel) }

$$\left(\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\ i : I \bullet pcs_i := 0); \\
\quad \mu\, X \bullet \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ m \rightarrow\ (\interleave\ i : I \bullet pcs_i := 1);\ X \\
\qquad \Box \\
\qquad \Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}\right) \setminus \{\!|\ m\ |\!\}$$

The operation of hiding the event m can be distributed over the variable declaration, the sequential composition, the iterated interleaving, the assignment and the fixed point operator.

= { Laws B.22, B.23, B.24, B.25 and B.26 (Hiding distribution) }

$$\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad \interleave\ i : I \bullet pcs_i := 0; \\
\quad \mu\, X \bullet \left(\begin{array}{l}
(\forall\, i : I \bullet pcs_i = 0)\ \&\ m \rightarrow\ (\interleave\ i : I \bullet pcs_i := 1);\ X \\
\Box \\
\Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}\right) \setminus \{\!|\ m\ |\!\}
\end{array}$$

Now we transform the external choice composed by the last two branches into a guarded action, so that the specification is in an adequate format for the application of the next law, which distributes the hiding over the external choice. Note that the guard of an iterated external choice is the existential quantification of their guards.

= { Law B.77 (External choice/Guarded action) }

$$
\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad \interleave\ i : I \bullet pcs_i := 0; \\
\quad \mu X \bullet \left(\begin{array}{l}
(\forall i : I \bullet pcs_i = 0)\ \&\ m \rightarrow\ (\interleave\ i : I \bullet pcs_i := 1);\ X \\
\Box \\
(\exists i : I \bullet pcs_i = 0) \vee (\exists i : I \bullet pcs_i = 1)\ \& \\
\qquad \Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}\right) \setminus \{\!|\ m\ |\!\}
\end{array}
$$

Now we can apply a law to distribute the hiding through the action system. Note that the action that follows the prefixing of m, and the action in the second branch of the outermost external choice do not refer to m; so hiding m is vacuous.

= { Laws B.19 (Hiding conditional external choice distribution 2a) and B.21 (Hiding identity) }

$$
\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\ i : I \bullet pcs_i := 0); \\
\quad \mu X \bullet \\
\qquad (\forall i : I \bullet pcs_i = 0)\ \&\ (\\
\qquad\quad (\interleave\ i : I \bullet pcs_i := 1) \\
\qquad\quad \Box \\
\qquad\qquad Stop \\
\qquad\qquad \sqcap \\
\qquad\qquad (\exists i : I \bullet pcs_i = 0) \vee (\exists i : I \bullet pcs_i = 1)\ \& \\
\qquad\qquad\quad \Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad \Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \\
\qquad) \\
\qquad \Box\ \neg\ (\forall i : I \bullet pcs_i = 0) \wedge ((\exists i : I \bullet pcs_i = 0) \vee (\exists i : I \bullet pcs_i = 1))\ \& \\
\qquad\quad \left(\begin{array}{l}
\Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}\right)
\end{array}
$$

We use a law to manipulate the guard of the second branch of the outermost external choice.

= { Law B.65 (Guard combination) }

```
var $pcs_i : \mathbb{N} \bullet$
    $(\interleave\, i : I \bullet pcs_i := 0);$
    $\mu\, X \bullet$
        $(\forall\, i : I \bullet pcs_i = 0)\ \&\ ($
            $(\interleave\, i : I \bullet pcs_i := 1)$
            $\Box$
                $Stop$
                $\sqcap$
                $(\exists\, i : I \bullet pcs_i = 0) \vee (\exists\, i : I \bullet pcs_i = 1)\ \&$
                    $\Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X$
                    $\Box$
                    $\Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X$
        $)$
        $\Box\ \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \&$
            $\left(\begin{array}{l} (\exists\, i : I \bullet pcs_i = 0) \vee (\exists\, i : I \bullet pcs_i = 1)\ \& \\ \quad \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\ \quad \Box \\ \quad \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \end{array}\right)$
```

Now we transform the guarded external choices back to a simple external choice again.

= { Law B.77 (External choice/Guarded action) }

```
var $pcs_i : \mathbb{N} \bullet$
    $(\interleave\, i : I \bullet pcs_i := 0);$
    $\mu\, X \bullet$
        $(\forall\, i : I \bullet pcs_i = 0)\ \&\ ($
            $(\interleave\, i : I \bullet pcs_i := 1)$
            $\Box$
                $Stop$
                $\sqcap$
                $\left(\begin{array}{l} \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \end{array}\right)$
        $)$
        $\Box\ \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \&\ \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X$
                                          $\Box$
                                          $\Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X$
```

We distribute the external choice over the internal choice, in the first branch of the outermost external choice.

= { Law B.75 (External choice/Internal choice - distribution) }

$$
\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\, i : I \bullet pcs_i := 0); \\
\quad \mu\, X \bullet \\
\quad\quad (\forall\, i : I \bullet pcs_i = 0)\ \&\ (\\
\quad\quad\quad \left(\begin{array}{l} (\interleave\, i : I \bullet pcs_i := 1);\ X \\ \Box \\ Stop \end{array} \right) \\
\quad\quad\quad \sqcap \\
\quad\quad\quad \left(\begin{array}{l} (\interleave\, i : I \bullet pcs_i := 1);\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \end{array} \right) \\
\quad\quad\quad) \\
\quad\quad \Box\ \neg\,(\forall\, i : I \bullet pcs_i = 0)\ \&\ \begin{array}{l} \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \end{array}
\end{array}
$$

The action *Stop* is the unit of the external choice, so our model can be further simplified.

$= \{$ Law B.76 (External choice unit) $\}$

$$
\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\, i : I \bullet pcs_i := 0); \\
\quad \mu\, X \bullet \\
\quad\quad (\forall\, i : I \bullet pcs_i = 0)\ \&\ (\\
\quad\quad\quad (\interleave\, i : I \bullet pcs_i := 1);\ X \\
\quad\quad\quad \sqcap \\
\quad\quad\quad \left(\begin{array}{l} (\interleave\, i : I \bullet pcs_i := 1);\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \end{array} \right) \\
\quad\quad\quad) \\
\quad\quad \Box\ \neg\,(\forall\, i : I \bullet pcs_i = 0)\ \&\ \begin{array}{l} \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\ \Box \\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \end{array}
\end{array}
$$

Note that we have an internal choice between the internal action $\interleave\, i : I \bullet pcs_i := 1$ and an external choice that offers as choice that same internal action. This is equivalent to keeping only the external choice.

$= \{$ Law B.13 (External choice/Internal choice - Internal action) $\}$

$$\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\, i : I \bullet pcs_i := 0); \\
\quad \mu\, X \bullet \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ \left(\begin{array}{l}
(\interleave\, i : I \bullet pcs_i := 1);\ X \\
\Box \\
\Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}\right) \\
\qquad \Box\ \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \&\ \begin{array}{l}
\Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}
\end{array}$$

We now distribute the guards over the external choice.

= { Law B.67 (Guard/External choice - distribution) }

$$\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\, i : I \bullet pcs_i := 0); \\
\quad \mu\, X \bullet \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ (\interleave\, i : I \bullet pcs_i := 1);\ X \\
\qquad \Box \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \&\ \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \&\ \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}$$

We note that there are some common branches in the external choice. We can combine these branches with the disjunction of guards.

= { Law B.66 (Guard expansion) }

$$\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\, i : I \bullet pcs_i := 0); \\
\quad \mu\, X \bullet \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ (\interleave\, i : I \bullet pcs_i := 1);\ X \\
\qquad \Box \\
\qquad (\forall\, i : I \bullet pcs_i = 0) \lor \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \& \\
\qquad\quad \Box\, i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall\, i : I \bullet pcs_i = 0) \lor \neg\ (\forall\, i : I \bullet pcs_i = 0)\ \& \\
\qquad\quad \Box\, i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}$$

Also note that $(\forall\, i : I \bullet pcs_i = 0) \lor \neg\ (\forall\, i : I \bullet pcs_i = 0)$ is always true.

= { Predicate calculus }

$$\begin{array}{l}
\textbf{var } pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\, i : I \bullet pcs_i := 0); \\
\quad \mu X \bullet \\
\qquad (\forall i : I \bullet pcs_i = 0) \,\&\, (\interleave\, i : I \bullet pcs_i := 1);\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet pcs_i = 0 \,\&\, interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet pcs_i = 1 \,\&\, q.i \rightarrow\ pcs_i := 0;\ X
\end{array}$$

To finalise, we substitute the interleaving of internal actions in the first branch of the external choice by a sequential composition of internal actions.

= { Law B.17 (Interleaving/Sequential composition - internal actions 2) }

$$\begin{array}{ll}
\textbf{var } pcs_i : \mathbb{N} \bullet & \\
\quad (\interleave\, i : I \bullet pcs_i := 0); & \\
\quad \mu X \bullet & \\
\qquad (\forall i : I \bullet pcs_i = 0) \,\&\, (;\ i : I \bullet pcs_i := 1);\ X & \text{(I.a)} \\
\qquad \Box & \\
\qquad \Box\, i : I \bullet pcs_i = 0 \,\&\, interrupt.i \rightarrow\ pcs_i := 0;\ X & \text{(I.b)} \\
\qquad \Box & \\
\qquad \Box\, i : I \bullet pcs_i = 1 \,\&\, q.i \rightarrow\ pcs_i := 0;\ X & \text{(I.c)}
\end{array}$$

In this resulting model, we have eliminated the parallelism of processes and the multi-synchronised event m. We give an intuition for why this model corresponds to the original one.

The array of variables pcs, indexed by I, serves to control the state of each parallel process $i \in I$ from the original model. The possible states are: zero, where the process can choose between the events m and $interrupt.i$; and 1, where the event m has just occurred.

When a process is in state zero, it can perform $interrupt.i$, independently (note that (I.b) is a multiple external choice), or m (I.a). Since m is shared by all processes, they all must be in state zero, in order that this event can occur. This is what the guard of (I.a) states. After the multi-synchronisation, which is implicit in the resulting model, all processes go to state 1. Once in this state, they must perform $q.i$ before going back to initial state (I.c), where they are able to execute the other events again.

6.3.2 Transforming *Network_Impl* into an action system

In this section, we transform the model for the implementation into an action system. The refinement steps are quite similar to the ones presented in the last section.

process *Network_Impl* $\widehat{=}$

$$
\left(\begin{array}{l}
|||\, i : I \bullet \\
\quad \mu X \bullet to_A!i \rightarrow \\
\qquad\qquad from_A.i?any \rightarrow \\
\qquad\qquad\quad to_B!i \rightarrow \\
\qquad\qquad\qquad from_B.i?synchronised \rightarrow \\
\qquad\qquad\qquad\quad (synchronised = true)\ \&\ q.i \rightarrow X \\
\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\quad (synchronised = false)\ \&\ X \\
\qquad\qquad \Box \\
\qquad\qquad interrupt.i \rightarrow \\
\qquad\qquad\quad to_A!(flip\ i) \rightarrow X \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad from_A.i?anyt \rightarrow to_B!(flip\ i) \rightarrow from_B.i?any \rightarrow X
\end{array}\right) \quad \textsf{(II.a)}
$$

$$
[\![\{ to_A, from_A, to_B, from_B \}]\!]
$$

$$
\left(\begin{array}{l}
(\mu X \bullet count : I \bullet \\
\quad (count > 0 \wedge count \leq n)\ \& \\
\qquad to_A?nextOffer \rightarrow \\
\qquad\quad (nextOffer \geq 0)\ \&\ X(count - 1) \\
\qquad\quad \Box \\
\qquad\quad (nextOffer < 0)\ \&\ X(count + 1) \\
\quad \Box \\
\quad (count = 0)\ \& \\
\qquad (\mu Y \bullet i : I \bullet \\
\qquad\quad (i < n)\ \&\ from_A.i!true \rightarrow Y(i + 1) \\
\qquad\quad \Box \\
\qquad\quad (i = n)\ \& \\
\qquad\qquad (\mu Z \bullet i, count : I \bullet \\
\qquad\qquad\quad (i \geq 0 \wedge i < n)\ \&\ to_B?nextcommit \rightarrow \\
\qquad\qquad\qquad (nextcommit \geq 0)\ \&\ Z(i + 1, count - 1) \\
\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad (nextcommit < 0)\ \&\ Z(i + 1, count) \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad (i = n)\ \& \\
\qquad\qquad\qquad (\mu W \widehat{=} i : I \bullet \\
\qquad\qquad\qquad\quad (i < n)\ \&\ from_B.i!(count = 0) \rightarrow W(i + 1) \\
\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\quad (i = n)\ \&\ X(n) \\
\qquad\qquad\qquad)(0) \\
\qquad\qquad)(0, n) \\
\qquad)(0) \\
)(n)
\end{array}\right) \quad \textsf{(II.b)}
$$

First, let us consider the interleaving of clients (II.a). We transform each interleaved client into an action system. The result is similar to that of Law B.1 (Action system conversion): we have the program counter *pc*, but this time there are eight possible states for the action system. The derivation of Lemma B.98 is in Appendix C of the extended version. It includes the application of the Least Fixed Point Law, used to refine a recursion.

(II.a)

= { Lemma B.98 }

```
$\interleave\ i : I \bullet$
    $\mathbf{var}\ pc : \mathbb{N}, sync : Boolean \bullet$
        $pc := 0;$
        $\mu X \bullet$
            $pc = 0 \ \&\ to_A!i \rightarrow pc := 1;\ X$
            $\Box\ pc = 1 \ \&\ from_A.i?any \rightarrow pc := 2;\ X$
            $\Box\ pc = 2 \ \&\ to_B!i \rightarrow pc := 3;\ X$
            $\Box\ pc = 3 \ \&\ from_B.i?synchronised \rightarrow pc := 4;\ sync := synchronised;\ X$
            $\Box\ (pc = 4 \wedge sync = true) \ \&\ q.i \rightarrow pc := 0;\ X$
            $\Box\ (pc = 4 \wedge sync = false) \ \&\ pc := 0;\ X$
            $\Box\ pc = 1 \ \&\ interrupt.i \rightarrow pc := 5;\ X$
            $\Box\ pc = 5 \ \&\ to_A!(flip\ i) \rightarrow pc := 0;\ X$
            $\Box\ pc = 5 \ \&\ from_A.i?anyt \rightarrow pc := 7;\ X$
            $\Box\ pc = 7 \ \&\ to_B?(flip\ i) \rightarrow pc := 8;\ X$
            $\Box\ pc = 8 \ \&\ from_B.i?anyf \rightarrow pc := 0;\ X$
```

As it was done with the model of the specification, we can apply laws to transform the interleaving of action systems into a single action system. As we widen the scope of the counter *pc*, we have an array of counters pc_i, instead. We then get the following action system for the interleaving of clients.

= { Laws B.88 (Parallel state), B.89 (Parallel assignment) and B.2 (Action system parallel)}

```
$\mathbf{var}\ pc_i : \mathbb{N}, sync_i : Boolean \bullet$
    $\interleave\ i : I \bullet pc_i := 0;$
    $\mu X \bullet$
        $\Box\ i : I \bullet pc_i = 0 \ \&\ to_A!i \rightarrow pc_i := 1;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 1 \ \&\ from_A.i?any \rightarrow pc_i := 2;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 2 \ \&\ to_B!i \rightarrow pc_i := 3;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 3 \ \&\ from_B.i?synchronised \rightarrow pc_i := 4;\ sync_i := synchronised;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 4 \wedge sync_i = true) \ \&\ q.i \rightarrow pc_i := 0;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 4 \wedge sync_i = false) \ \&\ pc_i := 0;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 1 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 5 \ \&\ to_A!(flip\ i) \rightarrow pc_i := 0;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 5 \ \&\ from_A.i?anyt \rightarrow pc_i := 7;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 7 \ \&\ to_B?(flip\ i) \rightarrow pc_i := 8;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 8 \ \&\ from_B.i?anyf \rightarrow pc_i := 0;\ X$
```

Now we transform the controller into an action system, using Lemma B.99. The variable *pc* is the program counter for this action system.

The proof of this lemma is also in Appendix C of the extended version. The controller is a parametrised recursion, so, the first step in is to transform it into a non-parametrised recursion. For that, we need to convert the parameters into local variables, and rename them where necessary, to avoid confusion; this originate variables $count_x$, $count_z$, i_y, i_z, i_w. After that, the steps are pretty similar to those of Lemma B.98 for the conversion of clients; we use Laws B.33 (Least fixed point) and B.29 (Assumption/recursion - refinement) to prove refinement in both directions.

(II.b)

= { Lemma B.99 }

```
$\textbf{var}\ pc : \mathbb{N};\ count_x, count_z, i_y, i_z, i_w : I;\ nextO, nextC : Message \bullet$
    $pc := 0;$
    $count_x := n;$
    $\mu X \bullet$
        $(pc = 0 \land count_x > 0)\ \&\ to_A?nextOffer \rightarrow pc := 1;\ nextO := nextOffer;\ X$
        $\Box$
        $(pc = 1 \land nextO \geq 0)\ \&\ pc := 0;\ count_x := count_x - 1;\ X$
        $\Box$
        $(pc = 1 \land nextO < 0)\ \&\ pc := 0;\ count_x := count_x + 1;\ X$
        $\Box$
        $(pc = 0 \land count_x = 0)\ \&\ pc := 2;\ i_y := 0;\ X$
        $\Box$
        $(pc = 2 \land i_y < n)\ \&\ from_A.i_y!true \rightarrow pc := 2;\ i_y := i_y + 1;\ X$
        $\Box$
        $(pc = 2 \land i_y = n)\ \&\ pc := 3;\ i_z := 0;\ count_z := n;\ X$
        $\Box$
        $(pc = 3 \land i_z \geq 0 \land i_z < n)\ \&\ to_B?nextCommit \rightarrow pc := 4;\ nextC := nextCommit;\ X$
        $\Box$
        $(pc = 4 \land nextC \geq 0)\ \&\ pc := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X$
        $\Box$
        $(pc = 4 \land next < 0)\ \&\ pc := 3;\ i_z := i_z + 1;\ X$
        $\Box$
        $(pc = 3 \land i_z = n)\ \&\ pc := 5;\ i_w = 0;\ X$
        $\Box$
        $(pc = 5 \land i_w < n)\ \&\ from_B.i_w!(count_z = 0) \rightarrow pc := 5;\ i_w := i_w + 1;\ X$
        $\Box$
        $(pc = 5 \land i_w = n)\ \&\ pc := 0;\ X$
```

Now we merge the action systems for the interleaving of clients and the controller. To do this, we combine the guards of any events on which the parallel processes synchronise using Law B.3 (Merge action system parallel); and we also compose in parallel the actions that follow the prefixing. Since these actions are only internal and do not share the same variables, their parallelism is equal to their sequential composition (Laws B.91- Parallelism/Interleaving - equivalence 2 and B.16 - Interleaving/Sequential composition - internal actions).

= { Laws B.3, B.91 and B.16 }

```
**var** $pc : \mathbb{N};\ count_x, count_z, i_y, i_z, i_w : I;\ nextO, nextC : Message;\ pc_i : \mathbb{N};\ sync_i : Boolean \bullet$
    $pc := 0;$
    $count_x := n;$
    $(|||\ i : I \bullet pc_i := 0);$
    $\mu X \bullet$
        $\Box\ i : I \bullet (pc_i = 0 \land pc = 0 \land count_x > 0)\ \&\ to_A.i \rightarrow$
            $pc_i := 1;\ pc := 1;\ nextO := i;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 1 \land pc = 2 \land i_y < n \land i = i_y)\ \&\ from_A.i.true \rightarrow$
            $pc_i := 2;\ pc := 2;\ i_y := i_y + 1;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 2 \land pc = 3 \land i_z \geq 0 \land i_z < n)\ \&\ to_A.i \rightarrow$
            $pc_i := 3;\ pc := 4;\ nextC := i;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 3 \land pc = 5 \land i_w < n \land i = i_w)\ \&\ from_A.i.(count_z = 0) \rightarrow$
            $pc_i := 4;\ pc := 5;\ sync_i := (count_z = 0);\ i_w := i_w + 1;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 5 \land pc = 0 \land count_x > 0)\ \&\ to_A.(flip\ i) \rightarrow$
            $pc_i := 0;\ pc := 1;\ nextO := flip\ i;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 5 \land pc = 2 \land i_y < n \land i = i_y)\ \&\ from_A.i.true \rightarrow$
            $pc_i := 7;\ pc := 2;\ i_y := i_y + 1;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 7 \land pc = 3 \land i_z \geq 0 \land i_z < n)\ \&\ to_A.(flip\ i) \rightarrow$
            $pc_i := 8;\ pc := 4;\ nextC := (flip\ i);\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 8 \land pc = 5 \land i_w < n \land i = i_w)\ \&\ from_A.i.(count_z = 0) \rightarrow$
            $pc_i := 0;\ pc := 5;\ i_w := i_w + 1;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 4 \land sync_i = true)\ \&\ q.i \rightarrow pc_i := 0;\ X$
        $\Box$
        $\Box\ i : I \bullet (pc_i = 4 \land sync_i = false)\ \&\ pc_i := 0;\ X$
        $\Box$
        $\Box\ i : I \bullet pc_i = 1\ \&\ interrupt.i \rightarrow pc_i := 5;\ X$
        $\Box\ (pc = 1 \land nextO \geq 0)\ \&\ pc := 0;\ count_x := count_x - 1;\ X$
        $\Box\ (pc = 1 \land nextO < 0)\ \&\ pc := 0;\ count_x := count_x + 1;\ X$
        $\Box\ (pc = 0 \land count_x = 0)\ \&\ pc := 2;\ i_y := 0;\ X$
        $\Box\ (pc = 2 \land i_y = n)\ \&\ pc := 3;\ i_z := 0;\ count_z := n;\ X$
        $\Box\ (pc = 4 \land nextC \geq 0)\ \&\ pc := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X$
        $\Box\ (pc = 4 \land next < 0)\ \&\ pc := 3;\ i_z := i_z + 1;\ X$
        $\Box\ (pc = 3 \land i_z = n)\ \&\ pc := 5;\ i_w = 0;\ X$
        $\Box\ (pc = 5 \land i_w = n)\ \&\ pc := 0;\ X$
$)\ \setminus\ \{to_A, from_A, to_B, from_B\}$
```

Our next step is to eliminate the communications on channels *to* and *from*, and the hiding operator. We use steps similar to those we used for the model of the specification. First, we distribute the hiding over the action system variable declarations, assignments and recursion (Laws B.22, B.23, B.24, B.25 and B.26 - Hiding distribution). Then we apply Law B.20 (Hiding conditional external choice 3) to distribute the guard over the action system. The result is not an action system anymore, but then we can manipulate guards and choices to make it an action system again. We are left with an action system similar to that obtained from the previous step, but without the communications. In the extended version of this thesis, the detailed steps of transformation are presented. Figure 6.3 shows the resulting model, where we can distinguish three groups of actions in the external choice: the first are the actions performed only by the clients; the second one are the communications between the controller and the clients, which now are implicit; the third are the actions performed only by the controller. This observation will guide us in the following steps of transformation.

$$
\begin{array}{l}
\mathbf{var}\ pc : \mathbb{N};\ count_x, count_z, i_y, i_z, i_w : I;\ nextO, nextC : Message;\ pc_i : \mathbb{N};\ sync_i : Boolean \bullet \\
\quad pc := 0; \\
\quad count_x := n; \\
\quad (\interleave\ i : I \bullet pc_i := 0); \\
\quad \mu X \bullet \\
\quad \left(\begin{array}{l}
\Box\ i : I \bullet (pc_i = 4 \land sync_i = true)\ \&\ q.i \to pc_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 4 \land sync_i = false)\ \&\ pc_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet pc_i = 1\ \&\ interrupt.i \to pc_i := 5;\ X
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{ll}
\Box\ i : I \bullet (pc_i = 0 \land pc = 0 \land count_x > 0)\ \&\ pc_i := 1;\ pc := 1;\ nextO := i;\ X & \text{(III.a)} \\
\Box & \\
\Box\ i : I \bullet (pc_i = 1 \land pc = 2 \land i_y < n \land i = i_y)\ \&\ pc_i := 2;\ pc := 2;\ i_y := i_y + 1;\ X & \\
\Box & \\
\Box\ i : I \bullet (pc_i = 2 \land pc = 3 \land i_z \geq 0 \land i_z < n)\ \&\ pc_i := 3;\ pc := 4;\ nextC := i;\ X & \text{(III.b)} \\
\Box & \\
\Box\ i : I \bullet (pc_i = 3 \land pc = 5 \land i_w < n \land i = i_w)\ \& & \\
\qquad\qquad pc_i := 4;\ pc := 5;\ sync_i := (count_z = 0);\ i_w := i_w + 1;\ X & \\
\Box & \\
\Box\ i : I \bullet (pc_i = 5 \land pc = 0 \land count_x > 0)\ \&\ pc_i := 0;\ pc := 1;\ nextO := flip\ i;\ X & \text{(III.c)} \\
\Box & \\
\Box\ i : I \bullet (pc_i = 5 \land pc = 2 \land i_y < n \land i = i_y)\ \&\ pc_i := 7;\ pc := 2;\ i_y := i_y + 1;\ X & \\
\Box & \\
\Box\ i : I \bullet (pc_i = 7 \land pc = 3 \land i_z \geq 0 \land i_z < n)\ \& & \\
\qquad\qquad pc_i := 8;\ pc := 4;\ nextC := (flip\ i);\ X & \text{(III.d)} \\
\Box & \\
\Box\ i : I \bullet (pc_i = 8 \land pc = 5 \land i_w < n \land i = i_w)\ \&\ pc_i := 0;\ pc := 5;\ i_w := i_w + 1;\ X &
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{ll}
(pc = 1 \land nextO \geq 0)\ \&\ pc := 0;\ count_x := count_x - 1;\ X & \text{(III.a}')\\
\Box\ (pc = 1 \land nextO < 0)\ \&\ pc := 0;\ count_x := count_x + 1;\ X & \text{(III.c}')\\
\Box\ (pc = 0 \land count_x = 0)\ \&\ pc := 2;\ i_y := 0;\ X & \\
\Box\ (pc = 2 \land i_y = n)\ \&\ pc := 3;\ i_z := 0;\ count_z := n;\ X & \\
\Box\ (pc = 4 \land nextC \geq 0)\ \&\ pc := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X & \text{(III.b}')\\
\Box\ (pc = 4 \land nextC < 0)\ \&\ pc := 3;\ i_z := i_z + 1;\ X & \text{(III.d}')\\
\Box\ (pc = 3 \land i_z = n)\ \&\ pc := 5;\ i_w = 0;\ X & \\
\Box\ (pc = 5 \land i_w = n)\ \&\ pc := 0;\ X &
\end{array}\right)
\end{array}
$$

Figure 6.3: Action system for *Network_Impl*

6.3.3 Proof of equality of the action systems for specification and implementation

In this section we start from the action system obtained from the implementation, and simplify it, until we reach the action system obtained from the specification. We have eliminated explicitly the hidden channels in both models, so that now we have the same interface (channels) in both models. However, each model uses different local variables. Therefore, to carry out the transformations we need to perform a data refinement. When performing data refinement in *Circus*, the structure of the specification is usually maintained.

In our approach for the transformation, we gradually simplify the action system for the implementation. The goal is to reach the same structure of the action system for the specification. Many of the steps involve merging branches or narrowing the set of values that the variables can represent. During the process of simplification, we sometimes make some of the variables useless, and eliminate them, sometimes using data refinement, sometimes using laws for manipulation of variables. When we reach the same structure of the action system for the specification, we are left with only the variables pc_i and $sync_i$. We apply data refinement to reach the model which uses only the variables pcs_i.

To proceed with the transformation, we note that it is possible to merge branches (III.a′), (III.b′), (III.c′) and (III.d′) with (III.a), (III.b), (III.c) and (III.d), respectively, using Law B.14 (Elimination of internal action 2). Intuitively, we can do that because the internal action in the first group enables the guard of the internal action in the second group, so we know that they will eventually occur.

$= \{$ Law B.14 $\}$

$$
\begin{array}{l}
\mathbf{var}\ pc : \mathbb{N};\ count_x, count_z, i_y, i_z, i_w : I;\ nextO, nextC : Message;\ pc_i : \mathbb{N};\ sync_i : Boolean \bullet \\
\quad pc := 0; \\
\quad count_x := n; \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu\, X \bullet \\
\quad \left(\begin{array}{l}
\Box\ i : I \bullet pc_i = 1\ \&\ interrupt.i \to pc_i := 5;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 4 \land sync_i = true)\ \&\ q.i \to pc_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 4 \land sync_i = false)\ \&\ pc_i := 0;\ X
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
\Box\ i : I \bullet (pc_i = 0 \land pc = 0 \land count_x > 0)\ \& \\
\qquad\qquad pc_i := 1;\ pc := 0;\ count_x := count_x - 1;\ nextO := i;\ X \qquad \text{(IV.a)} \\
\Box \\
\Box\ i : I \bullet (pc_i = 5 \land pc = 0 \land count_x > 0)\ \& \\
\qquad\qquad pc_i := 0;\ pc := 0;\ count_x := count_x + 1;\ nextO := \mathit{flip}\ i;\ X \qquad \text{(IV.b)} \\
\Box \\
\Box\ i : I \bullet (pc_i = 1 \land pc = 2 \land i_y < n)\ \&\ pc_i := 2;\ pc := 2;\ i_y := i_y + 1;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 5 \land pc = 2 \land i_y < n)\ \&\ pc_i := 7;\ pc := 2;\ i_y := i_y + 1;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 2 \land pc = 3 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 3;\ pc := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ nextC := i;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 7 \land pc = 3 \land i_z \geq 0 \land i_z < n)\ \&\ pc_i := 8;\ pc := 3; \\
\qquad\qquad i_z := i_z + 1;\ count_z := count_z - 1;\ nextC := (\mathit{flip}\ i);\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 3 \land pc = 5 \land i_w < n)\ \& \\
\qquad\qquad pc_i := 4;\ sync_i := (count_z = 0);\ pc := 5;\ i_w := i_w + 1;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 8 \land pc = 5 \land i_w < n)\ \&\ pc_i := 0;\ pc := 5;\ i_w := i_w + 1;\ X
\end{array}\right) \\
\quad \Box \\
\quad \left(\begin{array}{l}
(pc = 0 \land count_x = 0)\ \&\ pc := 2;\ i_y := 0;\ X \qquad \text{(IV.c)} \\
\Box\ (pc = 2 \land i_y = n)\ \&\ pc := 3;\ i_z := 0;\ count_z := n;\ X \\
\Box\ (pc = 3 \land i_z = n)\ \&\ pc := 5;\ i_w = 0;\ X \\
\Box\ (pc = 5 \land i_w = n)\ \&\ pc := 0;\ X
\end{array}\right)
\end{array}
$$

After the last transformation, variables *nextO* and *nextC* become useless, because their values are never used to determine the flow of execution of the action system. We can use Laws B.78 (Useless assignment - External choice) and B.30 (Useless assignment - Recursion) to move the assignments to these variables over the external choice and the recursion; then, we eliminate the assignments with Laws B.83 (Useless assignment) and B.84 (Useless assignment 2); and finally, since the variables are not used in the program, we can use Law B.87 (Unused variable) to eliminate the declarations of variables *nextO* and *nextC*. The result is a program similar to the previous one, but without the references to variables *nextO* and *nextC*.

Our next step is to eliminate the variable *pc*. This variable appeared when we the transformed the controller into an action system. It is the program counter for the action system that represented the controller. Its role is to control which phase of the protocol is being executed, so that only the guards for one phase of the protocol are enabled each time. After the last simplification, *pc* can assume four values: 0, 2, 3 or 5, each one corresponding to one phase of the protocol. Note that, in the previous specification, branches (IV.a), (IV.b) and (IV.c) are enabled only when *pc* is zero, and this corresponds to the first phase of the protocol. The guard of branch (IV.c) is the condition for finishing the first phase of the protocol. The corresponding action assigns the value 2 to the variable *pc*, and initiates the second phase of the protocol.

If we split up the action system into a sequential composition of four recursions, we can make the four phases of the protocol explicitly separated. This eliminates the need of the variable *pc*, since its role is only to determine which actions are enabled in each phase. The first law to apply is Law B.34 (Recursion split), to break the action system into a sequential composition of action systems. The result is an outer recursion, on S, which has as its body a sequential composition of recursions. Then we use Law B.27 (Recursion halt) to separate the action that follows the stopping condition in each recursion. The stopping condition is now followed by the action *Skip*, and the action is composed in sequence with the recursion.

To proceed, we use laws for manipulation of assumptions to eliminate the variable *pc*: we add assumptions after each assignment to the variable *pc* (Law B.53 - Assumption introduction - assignment), and place them just before the beginning of the next recursion (Law B.54 - Assumption move - assignment). Each of the recursions is now preceded by an assumption that states the initial value of *pc*. The assumptions make the assignments in the body of the recursions useless, so we can use Law B.36 (Useless variable - recursion) to eliminate them and the references to the variable *pc* in the guards.

When we have taken away all the references to the variable *pc* from inside the inner recursions, we are left with its initialisation and the assignments in between the recursions. The assignments to the variable *pc* are eliminated with Laws B.31 (Fixed Point Rolling), B.53 (Assumption introduction - assignment), B.48 (Move assignment), B.49 (Assignment sequence) and B.30 (Useless assignment - recursion). Finally, we eliminate the declaration of *pc* with Law B.87 (Unused variable).

= { Laws B.78, B.30, B.83, B.84, B.87, B.87, B.34, B.27, B.53, B.54, B.36, B.31, B.53, B.48, B.49, B.30 and B.87 }

$$
\begin{array}{l}
\mathbf{var}\ count_x := n, count_z, i_y, i_z, i_w, pc_i, sync_i \bullet \\
\quad (|||\ i : I \bullet pc_i := 0); \\
\quad \mu\, S \bullet \\
\quad\quad \mu\, X \bullet \\
\quad\quad \left(\begin{array}{l}
\Box\ i : I \bullet (pc_i = 0 \land count_x > 0)\ \&\ pc_i := 1;\ count_x := count_x - 1;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 5 \land count_x > 0)\ \&\ pc_i := 0;\ count_x := count_x + 1;\ X \\
\Box \\
(count_x = 0)\ \&\ \mathit{Skip} \\
\Box \\
\Box\ i : I \bullet pc_i = 1\ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 4 \land sync_i = true)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet (pc_i = 4 \land sync_i = false)\ \&\ pc_i := 0;\ X
\end{array}\right); \\
\quad\quad i_y := 0;
\end{array}
$$

$$
\begin{array}{l}
\mu X \bullet \\
\left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 1 \land i_y < n)\ \&\ pc_i := 2;\ i_y := i_y + 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 5 \land i_y < n)\ \&\ pc_i := 7;\ i_y := i_y + 1;\ X \\
\Box \\
(i_y = n)\ \&\ \mathit{Skip} \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ \mathit{interrupt}.i \rightarrow pc_i := 5;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land \mathit{sync}_i = \mathit{true})\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land \mathit{sync}_i = \mathit{false})\ \&\ pc_i := 0;\ X
\end{array}\right); \\
i_z := 0;\ \mathit{count}_z := n; \\
\mu X \bullet \\
\left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 2 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 3;\ i_z := i_z + 1;\ \mathit{count}_z := \mathit{count}_z - 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 7 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 8;\ i_z := i_z + 1;\ \mathit{count}_z := \mathit{count}_z - 1;\ X \\
\Box \\
(i_z = n)\ \&\ \mathit{Skip} \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ \mathit{interrupt}.i \rightarrow pc_i := 5;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land \mathit{sync}_i = \mathit{true})\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land \mathit{sync}_i = \mathit{false})\ \&\ pc_i := 0;\ X
\end{array}\right); \\
i_w = 0; \\
\mu X \bullet \\
\left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 3 \land i_w < n)\ \&\ pc_i := 4;\ \mathit{sync}_i := (\mathit{count}_z = 0);\ i_w := i_w + 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 8 \land i_w < n)\ \&\ pc_i := 0;\ i_w := i_w + 1;\ X \\
\Box \\
(i_w = n)\ \&\ \mathit{Skip} \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ \mathit{interrupt}.i \rightarrow pc_i := 5;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land \mathit{sync}_i = \mathit{true})\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land \mathit{sync}_i = \mathit{false})\ \&\ pc_i := 0;\ X
\end{array}\right); \\
\mathit{count}_x := n; \\
S
\end{array}
$$

Each of the inner recursions has an invariant; they are presented in Table 6.1. The invariants determine which are the possible states of a client in each phase of the protocol. We can use this observation to eliminate actions in the external choice that are never chosen. For that, we apply Law B.8 (Eliminate useless branch) to each recursion.

Recursion	**Invariant**
First recursion	$\forall\, i \bullet (pc_i = 0 \lor pc_i = 4 \lor pc_i = 1 \lor pc_i = 5)$
Second recursion	$\forall\, i \bullet (pc_i = 1 \lor pc_i = 2 \lor pc_i = 5 \lor pc_i = 7)$
Third recursion	$\forall\, i \bullet (pc_i = 2 \lor pc_i = 3 \lor pc_i = 7 \lor pc_i = 8)$
Fourth recursion	$\forall\, i \bullet (pc_i = 3 \lor pc_i = 8 \lor pc_i = 4 \lor pc_i = 0)$

Table 6.1: Recursion invariants

$= \{$ Law B.8 $\}$

$$
\begin{array}{l}
\mathbf{var}\ count_x := n, count_z, i_y, i_z, i_w, pc_i, sync_i \bullet \\
\quad (|||\, i : I \bullet pc_i := 0); \\
\quad \mu\, S \bullet \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 0 \land count_x > 0)\ \&\ pc_i := 1;\ count_x := count_x - 1;\ X \quad \text{(V.a)} \\
\Box \\
\Box\, i : I \bullet (pc_i = 5 \land count_x > 0)\ \&\ pc_i := 0;\ count_x := count_x + 1;\ X \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land sync_i = true)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land sync_i = false)\ \&\ pc_i := 0;\ X \\
\Box\ (count_x = 0)\ \&\ Skip
\end{array}\right); \\
\qquad i_y := 0; \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 1 \land i_y < n)\ \&\ pc_i := 2;\ i_y := i_y + 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 5 \land i_y < n)\ \&\ pc_i := 7;\ i_y := i_y + 1;\ X \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\Box\ (i_y = n)\ \&\ Skip
\end{array}\right); \\
\qquad i_z := 0;\ count_z := n; \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 2 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 7 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 8;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X \\
\Box\ (i_z = n)\ \&\ Skip
\end{array}\right); \\
\qquad i_w = 0; \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 3 \land i_w < n)\ \& \\
\qquad\qquad pc_i := 4;\ sync_i := (count_z = 0);\ i_w := i_w + 1;\ X \quad \text{(V.b)} \\
\Box \\
\Box\, i : I \bullet (pc_i = 8 \land i_w < n)\ \&\ pc_i := 0;\ i_w := i_w + 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land sync_i = true)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land sync_i = false)\ \&\ pc_i := 0;\ X \\
\Box\ (i_w = n)\ \&\ Skip
\end{array}\right); \\
\qquad count_x := n; \\
\qquad S
\end{array}
$$

Now we eliminate the variables $count_x$, which is only a counter, and $sync_i$. For this, we use data refinement with the following retrieve relation:

$$count_x = \#\{i \bullet pc_i = 0 \lor pc_i = 4\} \land (sync_i = true \Leftrightarrow count_z = 0)$$

The variable $count_x$ is used to count how many clients have not made the request for multi-synchronisation yet. It is initialised with n, the number of clients, and it is decremented when a client goes from state 0 to 1 (see action (V.a) of the previous program). The clients that have not made the request for multi-synchronisation are those that are in state 0 or 4. The variable $count_z$ counts the number of processes that chose to interrupt instead of participating in the multi-synchronisation. The variables $sync_i$ determine if the multi-synchronisation will happen or not. Each is assigned *true* if $count_z$ is zero (see branch (V.b) of the previous program).

The laws for simulation (Laws B.92, B.93, B.94, B.95, B.96 and B.97) justify this transformation.

= { Laws B.92, B.93, B.94, B.95, B.96 and B.97) }

$$
\begin{array}{l}
\mathbf{var}\ count_z, i_y, i_z, i_w, pc_i \bullet \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu S \bullet \\
\qquad \mu X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 0 \land \exists j \bullet (pc_j = 0 \lor pc_j = 4))\ \&\ pc_i := 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 5 \land \exists j \bullet (pc_j = 0 \lor pc_j = 4))\ \&\ pc_i := 0;\ X \\
\Box \\
(\forall i \bullet pc_i = 1 \lor pc_i = 5)\ \&\ Skip \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ interrupt.i \to pc_i := 5;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \to pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ X
\end{array}\right); \quad \text{(VI.a)} \\
\qquad i_y := 0; \\
\qquad \mu X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 1 \land i_y < n)\ \&\ pc_i := 2;\ i_y := i_y + 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 5 \land i_y < n)\ \&\ pc_i := 7;\ i_y := i_y + 1;\ X \\
\Box \\
(i_y = n)\ \&\ Skip \\
\Box \\
\Box\, i : I \bullet pc_i = 1\ \&\ interrupt.i \to pc_i := 5;\ X
\end{array}\right); \\
\qquad i_z := 0;\ count_z := n; \\
\qquad \mu X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 2 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 7 \land i_z \geq 0 \land i_z < n)\ \& \\
\qquad\qquad pc_i := 8;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X \\
\Box \\
(i_z = n)\ \&\ Skip
\end{array}\right);
\end{array}
$$

$$\begin{array}{l}
i_w = 0;\\
\mu\, X \bullet\\
\left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 3 \land i_w < n)\ \&\\
\qquad\qquad pc_i := 4;\ sync_i := (count_z = 0);\ i_w := i_w + 1;\ X\\
\Box\\
\Box\, i : I \bullet (pc_i = 8 \land i_w < n)\ \&\ pc_i := 0;\ i_w := i_w + 1;\ X\\
\Box\\
(i_w = n)\ \&\ Skip\\
\Box\\
\Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ X\\
\Box\\
\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ X
\end{array}\right);\\
count_x := n;\\
S
\end{array}$$

The stop condition of the first recursion states that no client can be in state zero when the recursion terminates. Based on this observation, we can apply a law that merges some branches of the action system and eliminates others. The result is shown next.

(VI.a)

= { Law B.9 (Merge branch 1)}

$$\begin{array}{l}
\mu\, X \bullet\\
\left(\begin{array}{l}
\Box\, i : I \bullet pc_i = 0\ \&\ pc_i := 1;\ X\\
\Box\\
\Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 0;\ X\\
\Box\\
\Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 5;\ X\\
\Box\\
\Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ X\\
\Box\\
\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ X
\end{array}\right)
\end{array}$$

The recursions can now be transformed to combinations of recursion and specification statements. At this point we eliminate the variables i_y, i_z, and i_w, because they are only counters for the previous recursions. The intuitive idea for this step is that we replace the internal actions with specification statements, and then we obtain a model that contains communications, and whose internal state transformations are described by specification statements. This is needed to simplify the model. Our goal is to minimise the number of branches of the action system that contains only internal transformations, so that we can have a system with a structure which is closer to the one we want to reach, that is, the action system for the specification. The laws that we used in this step are: Law B.15 (Elimination of internal action 2), for the first recursion; Law B.10 (Merge branches 2), for the second recursion; Law B.38 (Iteration), for the third recursion; and Law B.11 (Merge branches 3), for the fourth recursion.

After these transformations, the variables i_y, i_z, and i_w are not referenced anymore in the program. We use Law B.87 (Unused variable) to eliminate their declarations.

$= \{$ Laws B.15, B.10, B.38 and B.11 $\}$

$$
\begin{array}{l}
\textbf{var}\ pc_i, count_z \bullet \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu\, S \bullet \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet pc_i = 0 \,\&\, interrupt.i \to pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet pc_i = 0 \,\&\, interrupt.i \to pc_i := 5;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land count_z = 0) \,\&\, q.i \to pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0) \,\&\, pc_i := 0;\ X \\
\Box \\
(\forall\, i \bullet pc_i = 0 \lor pc_i = 5)\ \& \\
\qquad\qquad pc_i : [\forall\, i \bullet pc_i = 0 \lor pc_i = 5, \\
\qquad\qquad\qquad \forall\, i \bullet pc_i = 0 \Rightarrow pc_i' = 1 \land pc_i = 5 \Rightarrow pc_i' = pc_i]
\end{array}\right); \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet pc_i = 0 \,\&\, interrupt.i \to pc_i := 5;\ X \\
\Box \\
pc_i : [\forall\, i \bullet pc_i = 1 \lor pc_i = 5, \forall\, i \bullet pc_i = 1 \Rightarrow pc_i' = 2 \land pc_i = 5 \Rightarrow pc_i' = 7]
\end{array}\right); \\
\qquad pc_i, count_z : \\
\qquad\quad [\forall\, i \bullet pc_i = 2 \lor pc_i = 7, \\
\qquad\quad\ \forall\, i \bullet (pc_i = 2 \Rightarrow pc_i' = 3 \land pc_i = 7 \Rightarrow pc_i' = 8) \land count_z = \#\{i \bullet pc_i' = 8\}]; \\
\qquad pc_i : [\forall\, i \bullet pc_i = 3 \lor pc_i = 8, \forall\, i \bullet pc_i = 3 \Rightarrow pc_i' = 4 \land pc_i = 8 \Rightarrow pc_i' = 0]; \\
\qquad \mu\, X \bullet \\
\qquad \left(\begin{array}{l}
\Box\, i : I \bullet (pc_i = 4 \land count_z = 0) \,\&\, q.i \to pc_i := 0;\ X \\
\Box \\
\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0) \,\&\, pc_i := 0;\ X \\
\Box \\
Skip
\end{array}\right); \\
\qquad S
\end{array}
$$

Since the recursions are finite, for each one of them we can bring to the inside of the recursion the actions that follow the stop condition. For that, we use Law B.27 (Recursion halt). We rename the recursion variables to avoid confusion with their names.

$= \{$ Law B.27 $\}$

$$
\begin{array}{l}
\mathbf{var}\ pc_i, count_z \bullet \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu\, S \bullet \\
\qquad \mu\, X \bullet \\
\qquad\quad \Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad \Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\qquad\quad \Box \\
\qquad\quad \Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad \Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad (\forall\, i \bullet pc_i = 0 \lor pc_i = 5)\ \& \\
\qquad\qquad pc_i : [\forall\, i \bullet pc_i = 0 \lor pc_i = 5, \\
\qquad\qquad\qquad \forall\, i \bullet pc_i = 0 \Rightarrow pc_i' = 1 \land pc_i = 5 \Rightarrow pc_i' = pc_i]; \\
\\
\qquad\qquad \mu\, Y \bullet \\
\qquad\qquad\quad \Box\, i : I \bullet pc_i = 1\ \&\ interrupt.i \rightarrow pc_i := 5;\ Y \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad \left(
\begin{array}{l}
pc_i : [\forall\, i \bullet pc_i = 1 \lor pc_i = 5, \\
\qquad \forall\, i \bullet pc_i = 1 \Rightarrow pc_i' = 2 \land pc_i = 5 \Rightarrow pc_i' = 7]; \\
pc_i, count_z : [\forall\, i \bullet pc_i = 2 \lor pc_i = 7, \\
\qquad\qquad \forall\, i \bullet pc_i = 2 \Rightarrow pc_i' = 3 \land pc_i = 7 \Rightarrow pc_i' = 8 \land \\
\qquad\qquad\qquad count_z = \#\{i \bullet pc_i' = 8\}]; \\
pc_i : [\forall\, i \bullet pc_i = 3 \lor pc_i = 8, \\
\qquad \forall\, i \bullet pc_i = 3 \Rightarrow pc_i' = 4 \land pc_i = 8 \Rightarrow pc_i' = 0]; \\
\mu\, Z \bullet \\
\quad \Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ Z \\
\quad \Box \\
\quad \Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ Z \\
\quad \Box \\
\quad \mathit{Skip}
\end{array}
\right); \\
\qquad S
\end{array}
$$

We apply Law B.27 (Recursion halt) again to separate the action that follows the end of the recursion on Y, preparing for the next step.

= { Law B.27 }

$$
\begin{array}{l}
\textbf{var}\ pc_i, count_z \bullet \\
\quad \interleave\, i : I \bullet pc_i := 0; \\
\quad \mu\, S \bullet \\
\qquad \mu\, X \bullet \\
\qquad\quad \Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad \Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\qquad\quad \Box \\
\qquad\quad \Box\, i : I \bullet (pc_i = 4 \wedge count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad \Box\, i : I \bullet (pc_i = 4 \wedge count_z \neq 0) \ \&\ pc_i := 0;\ X \\
\qquad\quad \Box \\
\qquad\quad (\forall\, i \bullet pc_i = 0 \vee pc_i = 5) \ \& \\
\qquad\qquad pc_i : [\forall\, i \bullet pc_i = 0 \vee pc_i = 5, \\
\qquad\qquad\qquad \forall\, i \bullet pc_i = 0 \Rightarrow pc_i' = 1 \wedge pc_i = 5 \Rightarrow pc_i' = pc_i] \\
\qquad\qquad \mu\, Y \bullet \\
\qquad\qquad\quad \left(\begin{array}{l} \Box\, i : I \bullet pc_i = 1 \ \&\ interrupt.i \rightarrow pc_i := 5;\ Y \\ \Box \\ Skip \end{array} \right); \\
\qquad\qquad \left(\begin{array}{l}
pc_i : [\forall\, i \bullet pc_i = 1 \vee pc_i = 5, \\
\qquad \forall\, i \bullet pc_i = 1 \Rightarrow pc_i' = 2 \wedge pc_i = 5 \Rightarrow pc_i' = 7]; \\
pc_i, count_z : [\forall\, i \bullet pc_i = 2 \vee pc_i = 7, \\
\qquad\qquad \forall\, i \bullet pc_i = 2 \Rightarrow pc_i' = 3 \wedge pc_i = 7 \Rightarrow pc_i' = 8 \wedge \\
\qquad\qquad\qquad count_z = \#\{i \bullet pc_i' = 8\}]; \\
pc_i : [\forall\, i \bullet pc_i = 3 \vee pc_i = 8, \\
\qquad \forall\, i \bullet pc_i = 3 \Rightarrow pc_i' = 4 \wedge pc_i = 8 \Rightarrow pc_i' = 0]; \\
\mu\, Z \bullet \\
\quad \Box\, i : I \bullet (pc_i = 4 \wedge count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ Z \\
\quad \Box \\
\quad \Box\, i : I \bullet (pc_i = 4 \wedge count_z \neq 0) \ \&\ pc_i := 0;\ Z \\
\quad \Box \\
\quad Skip
\end{array} \right); \\
\qquad S
\end{array}
$$

With the assumption that the guard $(\forall\, i \bullet pc_i = 0 \vee pc_i = 5)$ introduces (Laws B.57 - Guard/assumption - introduction), we can swap the order of the specification statement and the recursion on Y (Law B.63 - Move specification statement), updating the guards in the recursion to check the old values of pc_i, that is, before the transformation that the specification statement describes. After that, we use Law B.27 (Recursion halt) to bring the actions that follows the recursion on Y back to the external choice again. The result is that we have a sequential composition of four specification statements following the recursion on Y.

= { Laws B.57, B.63 and B.27 }

```
var $pc_i, count_z$ •
    $\interleave\, i : I \bullet pc_i := 0;$
    $\mu\, S \bullet$
        $\mu\, X \bullet$
            $\Box\, i : I \bullet pc_i = 0 \;\&\; interrupt.i \rightarrow pc_i := 0;\; X$
            $\Box$
            $\Box\, i : I \bullet pc_i = 0 \;\&\; interrupt.i \rightarrow pc_i := 5;\; X$   (VII)
            $\Box$
            $\Box\, i : I \bullet (pc_i = 4 \land count_z = 0) \;\&\; q.i \rightarrow pc_i := 0;\; X$
            $\Box$
            $\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0) \;\&\; pc_i := 0;\; X$
            $\Box$
            $(\forall\, i \bullet pc_i = 0 \lor pc_i = 5) \;\&$
                $\mu\, Y \bullet$
                    $\Box\, i : I \bullet pc_i = 0 \;\&\; interrupt.i \rightarrow pc_i := 5;\; Y$   (VII)
                    $\Box$
                    (
                      $pc_i : [\forall\, i \bullet pc_i = 0 \lor pc_i = 5,$
                              $\forall\, i \bullet pc_i = 0 \Rightarrow pc_i' = 1 \land pc_i = 5 \Rightarrow pc_i' = pc_i]$
                      $pc_i : [\forall\, i \bullet pc_i = 1 \lor pc_i = 5,$
                              $\forall\, i \bullet pc_i = 1 \Rightarrow pc_i' = 2 \land pc_i = 5 \Rightarrow pc_i' = 7];$
                      $pc_i, count_z : [\forall\, i \bullet pc_i = 2 \lor pc_i = 7,$
                              $\forall\, i \bullet pc_i = 2 \Rightarrow pc_i' = 3 \land pc_i = 7 \Rightarrow pc_i' = 8 \land$
                                      $count_z = \#\{i \bullet pc_i' = 8\}];$
                      $pc_i : [\forall\, i \bullet pc_i = 3 \lor pc_i = 8,$
                              $\forall\, i \bullet pc_i = 3 \Rightarrow pc_i' = 4 \land pc_i = 8 \Rightarrow pc_i' = 0];$
                      $\mu\, Z \bullet$
                          $\Box\, i : I \bullet (pc_i = 4 \land count_z = 0) \;\&\; q.i \rightarrow pc_i := 0;\; Z$
                          $\Box$
                          $\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0) \;\&\; pc_i := 0;\; Z$
                          $\Box$
                          $Skip$
                    ) ;
        $S$
```

The branch (VII) appears in the recursion on X and also in the recursion on Y. This is a redundancy, since the decision to stop the first loop is non-deterministic. We can keep only the first one, using Law B.28 (Elimination of redundant branch in recursion).

The four specification statements transform, step by step, the variables pc_i with values 1 (the clients that did not interrupt) and 5 (the clients that interrupted), into 4 and 0, respectively. The third specification statement will also assign to $count_z$ the number of clients that did not interrupt. So we can merge these four specification statements into a single one, by applying Law B.42 (Sequential composition) repeatedly.

= { Laws B.28 and B.42 }

```
var $pc_i, count_z \bullet$
    $|||\ i : I \bullet pc_i := 0;$
    $\mu S \bullet$
        $\mu X \bullet$
            $\Box\ i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\ i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X$
            $\Box$
            $\Box\ i : I \bullet (pc_i = 4 \land count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\ i : I \bullet (pc_i = 4 \land count_z \neq 0) \ \&\ pc_i := 0;\ X$
            $\Box$
            $(\forall i \bullet pc_i = 0 \lor pc_i = 5) \ \&$
                $pc_i, count_z : [\forall i \bullet pc_i = 0 \lor pc_i = 5,$
                        $\forall i \bullet pc_i = 0 \Rightarrow pc_i' = 4 \land pc_i = 5 \Rightarrow pc_i' = 0 \land$
                                $count_z = \#\{i \bullet pc_i' = 0\}];$
                $\mu Z \bullet$
                    $\Box\ i : I \bullet (pc_i = 4 \land count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ Z$
                    $\Box$
                    $\Box\ i : I \bullet (pc_i = 4 \land count_z \neq 0) \ \&\ pc_i := 0;\ Z$
                    $\Box$
                    $Skip;$
        $S$
```

Again, there are redundant branches in the inner recursion. We can use Law B.28 (Elimination of redundant branch in recursion) to simplify the specification. After that, we are left with only the action *Skip* as the body of the recursion; the fixed point operator is clearly useless, and then can be eliminated. For that, we use Laws B.43 (Sequence unit) and B.37 (Useless recursion).

= { Laws B.28, B.43 and B.37 }

```
var $pc_i, count_z \bullet$
    $|||\ i : I \bullet pc_i := 0;$
    $\mu S \bullet$
        $\mu X \bullet$
            $\Box\ i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\ i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X$
            $\Box$
            $\Box\ i : I \bullet (pc_i = 4 \land count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\ i : I \bullet (pc_i = 4 \land count_z \neq 0) \ \&\ pc_i := 0;\ X$
            $\Box$
            $(\forall i \bullet pc_i = 0 \lor pc_i = 5) \ \&$
                $pc_i, count_z : [\forall i \bullet pc_i = 0 \lor pc_i = 5,$
                        $\forall i \bullet pc_i = 0 \Rightarrow pc_i' = 4 \land pc_i = 5 \Rightarrow pc_i' = 0 \land$
                                $count_z = \#\{i \bullet pc_i' = 0\}];$
        $S$
```

The outer recursion does nothing more than calling the inner recursion again when it finishes. It can be eliminated with the diagonal rule.

```
= { Law B.32 (Fixed point diagonal) }
    $\mathbf{var}\ pc_i, count_z \bullet$
        $||| \ i : I \bullet pc_i := 0;$
        $\mu X \bullet$
            $\Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X$
            $\Box$
            $\Box\, i : I \bullet (pc_i = 4 \land count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0) \ \&\ pc_i := 0;\ X$
            $\Box$
            $(\forall\, i \bullet pc_i = 0 \lor pc_i = 5) \ \&$
                $pc_i, count_z : [\forall\, i \bullet pc_i = 0 \lor pc_i = 5,$
                        $\forall\, i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land$
                                $count_z = \#\{i \bullet pc'_i = 0\}];\ X$
```

We introduce an alternation to split the last branch in two cases: the case where all pc_is are in stage zero, which means that there was no interruption and, thus, the synchronisation will happen; and the case where a interruption has happened. This is valid because the precondition implies that one of the guards is always true.

```
= { Law B.39 (Alternation introduction) }
    $\mathbf{var}\ pc_i, count_z \bullet$
        $||| \ i : I \bullet pc_i := 0;$
        $\mu X \bullet$
            $\Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X$
            $\Box$
            $\Box\, i : I \bullet (pc_i = 4 \land count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X$
            $\Box$
            $\Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0) \ \&\ pc_i := 0;\ X$
            $\Box$
            $(\forall\, i \bullet pc_i = 0 \lor pc_i = 5) \ \&$
                $\mathbf{if}\ \forall\, i \bullet pc_i = 0 \rightarrow$
                    $pc_i, count_z : [\forall\, i \bullet pc_i = 0 \lor pc_i = 5 \land \forall\, i \bullet pc_i = 0,$
                            $\forall\, i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land$
                                    $count_z = \#\{i \bullet pc'_i = 0\}];\ X$
                $[\!] \neg\, (\forall\, i \bullet pc_i = 0) \rightarrow$
                    $pc_i, count_z : [\forall\, i \bullet pc_i = 0 \lor pc_i = 5 \land \neg\, (\forall\, i \bullet pc_i = 0),$
                            $\forall\, i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land$
                                    $count_z = \#\{i \bullet pc'_i = 0\}];\ X$
                $\mathbf{fi}$
```

It is possible to transform the alternation to an external choice, since the guards exclude each other and one of them will always be true.

$= \{$ Law B.41 (Alternation/Guarded Actions - interchange) $\}$

$$\begin{array}{l}
\mathbf{var}\ pc_i, count_z \bullet \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu X \bullet \\
\qquad \Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall i \bullet pc_i = 0 \lor pc_i = 5)\ \& \\
\qquad\quad \forall i \bullet pc_i = 0\ \& \\
\qquad\qquad pc_i, count_z : [\forall i \bullet pc_i = 0 \lor pc_i = 5 \land \forall i \bullet pc_i = 0, \\
\qquad\qquad\qquad \forall i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land \\
\qquad\qquad\qquad\qquad count_z = \#\{i \bullet pc'_i = 0\}];\ X \\
\qquad\quad \Box \\
\qquad\quad \neg\, (\forall i \bullet pc_i = 0)\ \& \\
\qquad\qquad pc_i, count_z : [\forall i \bullet pc_i = 0 \lor pc_i = 5 \land \neg\, (\forall i \bullet pc_i = 0), \\
\qquad\qquad\qquad \forall i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land \\
\qquad\qquad\qquad\qquad count_z = \#\{i \bullet pc'_i = 0\}];\ X
\end{array}$$

We distribute the guards, and now we have a single external choice.

$= \{$ Law B.67 (Guard/External choice - distribution) $\}$

$$\begin{array}{l}
\mathbf{var}\ pc_i, count_z \bullet \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu X \bullet \\
\qquad \Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 5;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \land count_z \neq 0)\ \&\ pc_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall i \bullet pc_i = 0 \lor pc_i = 5) \land (\forall i \bullet pc_i = 0)\ \& \\
\qquad\quad pc_i, count_z : [\forall i \bullet pc_i = 0 \lor pc_i = 5 \land \forall i \bullet pc_i = 0, \\
\qquad\qquad \forall i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land \\
\qquad\qquad\qquad count'_z = \#\{i \bullet pc'_i = 0\}];\ X \\
\qquad \Box \\
\qquad (\forall i \bullet pc_i = 0 \lor pc_i = 5) \land \neg\, (\forall i \bullet pc_i = 0)\ \& \\
\qquad\quad pc_i, count_z : [\forall i \bullet pc_i = 0 \lor pc_i = 5 \land \neg\, (\forall i \bullet pc_i = 0), \\
\qquad\qquad \forall i \bullet pc_i = 0 \Rightarrow pc'_i = 4 \land pc_i = 5 \Rightarrow pc'_i = 0 \land \\
\qquad\qquad\qquad count'_z = \#\{i \bullet pc'_i = 0\}];\ X
\end{array}$$

We simplify the guards, the pre-condition and the pos-condition of the first specification statement. Law B.61 allows to substitute a pos-condition *pos* by pos_1, if, in the context of the pre-condition, *pos* and pos_1 are equivalent, which is the case here.

= { Predicate calculus, Law B.61 (Equivalent pos-condition) }

$$
\begin{array}{l}
\textbf{var}\ pc_i, count_z \bullet \\
\quad ||| \ i : I \bullet pc_i := 0; \\
\quad \mu X \bullet \\
\qquad \Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X \qquad \text{(VIII.a)} \\
\qquad \Box \\
\qquad \Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 5;\ X \qquad \text{(VIII.b)} \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \wedge count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X \qquad \text{(VIII.c)} \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \wedge count_z \neq 0) \ \&\ pc_i := 0;\ X \qquad \text{(VIII.d)} \\
\qquad \Box \\
\qquad (\forall\, i \bullet pc_i = 0) \ \& \\
\qquad\quad pc_i, count_z : [\forall\, i \bullet pc_i = 0, \\
\qquad\qquad \forall\, i \bullet pc_i' = 4 \wedge count_z' = 0];\ X \qquad \text{(VIII.e)} \\
\qquad \Box \\
\qquad (\forall\, i \bullet pc_i = 0 \vee pc_i = 5) \wedge \neg\, (\forall\, i \bullet pc_i = 0) \ \& \\
\qquad\quad pc_i, count_z : [\forall\, i \bullet pc_i = 0 \vee pc_i = 5 \wedge \neg\, (\forall\, i \bullet pc_i = 0), \\
\qquad\qquad \forall\, i \bullet pc_i = 0 \Rightarrow pc_i' = 4 \wedge pc_i = 5 \Rightarrow pc_i' = 0 \wedge \\
\qquad\qquad\qquad count_z' = \#\{i \bullet pc_i' = 0\}];\ X \qquad \text{(VIII.f)}
\end{array}
$$

The action system contains three irrelevant branches, whose behaviour can be simulated by the execution in sequence of other branches. We can give an informal explanation of why they are irrelevant. Since all the pc_is are initialised with zero, the branch (VIII.b) needs to be chosen at some point for the guard $(\forall\, i \bullet pc_i = 0 \vee pc_i = 5) \wedge \neg\, (\forall\, i \bullet pc_i = 0)$ of branch (VIII.f) to become available. If (VIII.f) is chosen, then those pc_is in stage 5 are reinitialised; and the branch (VIII.c) will eventually be chosen to reinitialise the pc_is that were in stage 0. For those branches that have performed an interruption when (VIII.f) is chosen, this is equivalent to branch (VIII.a), which performs an interruption and does not change state; for the others, this is all equivalent to *Skip*.

= { Law B.14 (Elimination of internal action) }

$$
\begin{array}{l}
\textbf{var}\ pc_i, count_z \bullet \\
\quad ||| \ i : I \bullet pc_i := 0; \\
\quad \mu X \bullet \\
\qquad \Box\, i : I \bullet pc_i = 0 \ \&\ interrupt.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\, i : I \bullet (pc_i = 4 \wedge count_z = 0) \ \&\ q.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall\, i \bullet pc_i = 0) \ \&\ pc_i, count_z : [\forall\, i \bullet pc_i = 0, \forall\, i \bullet pc_i' = 4 \wedge count_z' = 0];\ X
\end{array}
$$

The guard of the third branch matches the precondition of the specification statement. We can take advantage of that to insert an assumption and introduce an assignment.

= { Laws B.57 (Guard/Assumption - introduction), B.45 (Assignment) }

$$
\begin{array}{l}
\mathbf{var}\ pc_i, count_z \bullet \\
\quad \interleave\ i : I \bullet pc_i := 0; \\
\quad \mu X \bullet \\
\qquad \Box\ i : I \bullet pc_i = 0\ \&\ interrupt.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\ i : I \bullet (pc_i = 4 \land count_z = 0)\ \&\ q.i \rightarrow pc_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall\, i \bullet pc_i = 0)\ \&\ (;\ i : I \bullet pc_i := 4);\ count_z := 0;\ X
\end{array}
$$

And we can finally carry out a data refinement with a very straightforward retrieve relation:

$$(pcs_i = 1 \Leftrightarrow pc_i = 4 \land count_z = 0) \land (pcs_i = 0 \Leftrightarrow pc_i = 0)$$

The value zero for the program counter in both action systems (specification and implementation) represent the initial conditions. In the action system for the implementation, $pc_i = 4$ means that we are in the final stage of the protocol; $count_z = 0$ means that no client chose to interrupt. Therefore, the synchronisation occurs. This is equivalent, in the action system for the specification to the situation where a process is in state 1; this means that the multi-synchronisation has occurred. The laws for simulation are used to justify the data refinement.

= { Laws B.92, B.93, B.94, B.95, B.96 and B.97 (Simulation) }

$$
\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad \interleave\ i : I \bullet pcs_i := 0; \\
\quad \mu X \bullet \\
\qquad \Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow pcs_i := 0;\ X \\
\qquad \Box \\
\qquad \Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow pcs_i := 0;\ X \\
\qquad \Box \\
\qquad (\forall\, i : I \bullet pcs_i = 0)\ \&\ (;\ i : I \bullet pcs_i := 1);\ X
\end{array}
$$

We conclude with a reordering of the branches of the external choice.

= { Law B.72 (Commutativity of external choice) }

$$
\begin{array}{l}
\mathbf{var}\ pcs_i : \mathbb{N} \bullet \\
\quad (\interleave\ i : I \bullet pcs_i := 0); \\
\quad \mu X \bullet \\
\qquad \left(\begin{array}{l}
(\forall\, i : I \bullet pcs_i = 0)\ \&\ (;\ i : I \bullet pcs_i := 1);\ X \\
\Box \\
\Box\ i : I \bullet pcs_i = 0\ \&\ interrupt.i \rightarrow\ pcs_i := 0;\ X \\
\Box \\
\Box\ i : I \bullet pcs_i = 1\ \&\ q.i \rightarrow\ pcs_i := 0;\ X
\end{array}\right)
\end{array}
$$

And we finally reach the action system for the specification.

6.4 Final considerations

In this chapter we have verified the implementation for the multi-synchronisation protocol. We have proposed a specification for a situation where the multi-synchronised channel occurs in an external choice. The implementation is based on a protocol with four phases, in which a controller handles the requests for multi-synchronisation.

In [42] a simpler form of multi-synchronisation, which did not involve choice, was verified. This work was our major source of inspiration for the development of the refinement described in this chapter. We follow the same approach of reducing the program to an action system in the first stages of development. We used some laws introduced in that work, but we needed to propose some new laws.

Some of the new laws we have proposed are derived laws, that is, laws that result from the application of simpler laws in sequence. Some of the derivations are presented in the Appendix C of the extended version of this thesis [16].

Other laws are quite specific. Proof of refinement laws of a language involves the semantics of the language. *Circus* is based on the UTP, which is a theory of relations; refinement in the UTP is interpreted as inverse implication. To prove that $P_1 \sqsubseteq P_2$, we need to demonstrate that the semantic model for P_2 implies the semantic model for P_1. We left the proof of the refinement laws that we proposed as future work. In spite that, we believe that the refinement described in this work is an important step towards the verification of the multi-synchronisation protocol.

Chapter 7

Conclusions

In this chapter we draw conclusions and cite the main contributions of our work. We also present and analyse related work, and identify areas on which our work can be improved and extended.

7.1 Contributions

The use of tools is of fundamental importance to the practical application of formal methods. Theorem provers, model checkers, tools for refinement and translators between formalisms provide the necessary automation of techniques that help designers and developers to avoid errors and save time when manipulating the complex formulas that are typical of the use of formal methods.

In this work we have implemented a tool for the *Circus* language: *JCircus*, a translator from concrete *Circus* to Java. The implementation was based on previous work [29], which proposed rules for translation from *Circus* to Java. The implementation of *JCircus* showed that the automation of the translation strategy is feasible. Our effort for implementation revealed some errors in the original strategy, and alternative translation rules have been proposed.

The automation provided by *JCircus* protects the user from the tedious and error-prone task of manually coding the Java program resulting from the translation rules. Once the final concrete specification has been reached, an implementation can be obtained within minutes. *JCircus* is available for download from www.cs.york.ac.uk/circus, where we can also find the case studies that we have carried out.

The programs generated by *JCircus* also provide a graphical interface, so that users can interact with the program. Buttons represent channels, input parameters for the channels are entered on text fields, and output parameters can be seen on the screen. The graphical interface, however, is separated from the classes that implement each *Circus* program, so that it does not interfere on the translation rules. The interface helps the user to visualise what the program does and can be a useful tool to validate it. Therefore, *JCircus* proved itself not only an automatic translator, but also an automatic generator of animators for *Circus*.

We have followed a structured approach in the development of *JCircus*, and used UML to document the project. *JCircus* was implemented in Java, and uses the CZT framework, which was recently extended with support for *Circus*. The CZT framework was designed to provide support for Z and its extensions. Therefore, its design allows that extensions of formalisms can be easily included. This is an important point, as *Circus* itself has extensions under development, which include support for object-orientation, time and mobility. We want, in the future, to be able to extend *JCircus* for these extensions as well.

We have followed a test strategy that included various types of tests, and we can say that *JCircus* is quite well-tested. In total, we have tested about 1000 lines of LaTeX *Circus* specifications. These tests included simple test cases to test the correctness of implementation of each translation rule, and some simple programs like the GCD calculator described in Chapter 2.

The original translation strategy was presented in a didactic approach. First, it introduced rules that did not cover systems that contained multi-synchronisation or generic channels. Then, it was extended to consider these new features, and new versions of some rules were proposed to deal specifically with each of them. In order to implement a generic approach, we had to unify these different versions. This was specially important in the case of multi-synchronisation, because when translating a process, we do not know if their channels will take part or not in multi-synchronisation; this depends on which other process this process will be composed with. To solve this problem, we came up with a design that permitted the translation of channels that are at the same time generic and multi-synchronised, which the original strategy did not allow.

Our work also helped to make explicit some requirements that were implicit in the original strategy, like the requirement on hiding of channels, for example. This requirement determines that hiding of channels appears only as the last operator applied to a process in a process definition, and arises from the way hidden channels are treated by the strategy. The requirements are documented in Chapter 4.

The work on the formal verification of the multi-synchronisation was inspired on the results of [42] and also on a FDR script for verification of the multi-synchronisation protocol that is implemented by the tool. We wanted to verify the same protocol using refinement calculus, so that we have a more general result that is not limited to a fixed number of clients. We also have come up with new *Circus* refinement laws.

7.2 Related work

We have found some works that have been carried out on automatic translation of formalisms into some programming language. The main motivation for these works is to improve productivity and reliability: automation reduces the risk of human errors and saves human effort. Here we describe some of such works.

In [34], the authors present a tool that converts a subset of CSP into Handel-C code. Handel-C is a language designed to program hardware; its compiler generates scripts that can be used to program FPGAs. It has a syntax similar to the standard C, but provides built-in constructors to deal with concurrency and communication, such as channels, parallelism and alternation. The paper describes a methodology for implementing concurrent systems on FPGAs, starting from a CSP description. In this tool, CSP scripts are written in the machine-readable notation used by ProBE and FDR, with additional macro declarations for embedding Handel-C statements in the CSP script.

The work in [35] describes tools that automatically convert a subset of machine-readable CSP script to executable Java or C code with the use of library implementations of CSP. Namely, they use CCSP and CTJ; CCSP is a library for the C programming language and CTJ is also for Java, with support for real-time features. These tools also deal with specifications written in the machine-readable form and accepts a subset of CSP. However, the tools do not handle non-CSP code, that is, operations over variables and expressions like those available in *Circus*. The operators of CSP are also limited; replication operators, boolean guards and sequential composition are not covered.

As far as we know, these works described do not provide a formal specification of the translation rules. Also, we have not found references to concerns about formal verification of the strategy. Correctness of the implementations have been only evidenced by tests and simulations.

Both tools described were in initial stages of development, as ours is. The CSP/Handel-C translator serves a purpose different from ours. Their goal is to have a Handel-C script to program hardware, while ours is to develop software systems based on Java. Anyway, a distinguishing feature of *JCircus* is the graphical interface generated which provides friendly interaction with the user, and helps to visualise the behaviour of the resulting program.

In [12], a translation strategy from CSP-OZ to Java, using the CTJ library is presented. Since it deals with a combination of CSP with a state-based language (Object-Z), and not purely CSP, this is the closest work to ours. Also, the translation rules are described in a way similar to ours. In the CSP-OZ notation, the behavioural and communication aspects are kept separated from the state operations. The implementation reflects this architecture, and as a result, the code is usually inefficient and unnecessarily complicated. *Circus* is a programming as well as a specification language, in which CSP and Z constructs are freely mixed; actually, the difficulties found in this work were one of the motivations for the design of *Circus*. As a consequence of its simplicity, the translation rules for *Circus* result in less complicated programs.

In [27], there is an example of the formal development of a control system based on CSP: an abstract CSP specification is refined to a concrete CSP model, which is implemented using JCSP. As part of the refinement, multi-way synchronisation is eliminated. The protocol used and its proof of correctness are not presented, but the example indicates that the protocol is similar to that implemented in *JCircus*, which makes possible the elimination of several multi-synchronisations that take part in an external choice. It follows the ideas presented in [42]. Translation to JCSP seems to be a simpler matter in [27], since the starting point is a CSP model without multi-way synchronisation. In our tool, we automate that development step, using a JCSP implementation of a multi-way synchronisation in Java to allow the translation of more elaborate *Circus* programs. We have also verified a simpler version of the protocol, which allows at most one multi-synchronisation in a choice. This enhances our confidence in the translation; the protocol that is actually implemented is an extension, and can be verified using a similar approach.

7.3 Future work

The implementation of *JCircus* has proved itself an interesting work, but it is far from complete. There are still some features to implement, which can make the tool more robust and useful.

The most important extension is the provision of an implementation for more kinds of *Circus* types; so far, we have only implemented free types and $\mathbb{A}$ (number). It is important to deal with sets, cartesian products and schema types. The use of a broader range of types will require implementations for more types of Z expressions. An interesting extension is the implementation of the mathematical toolkit of the Standard Z in Java. So far, only the number toolkit has been implemented, to provide support for the translation of expressions of the built-in type $\mathbb{A}$. Some limitations of JCSP prevented the implementation of important features of *Circus*. For instance, there are limitations on the forms of parallelism and interleaving that can be handled. JCSP provides an implementation for Hoare's version of parallelism: the alphabets of the parallel processes are implicit and the processes synchronize on all events that they have in common. In JCSP, it is not possible to define the synchronisation set, as in Roscoe's and *Circus*' interface parallel. It also does not implement interleaving; this operation is implemented with the parallel operator, only in the case where the processes do not share any channel.

Another limitation of JCSP is that it does not support output channels in an alternation. Another piece of future work is the investigation of alternative implementations for the external choice, that do not use the `Alternative` class of JCSP, to allow the definition of output guards in an external choice. Another possibility is to find a protocol to eliminate output channels from alternations, in the same way that we used a protocol to eliminate multi-synchronisations.

Some constructs of the *Circus* grammar are not supported by our tool for the reason that we still do not have a robust enough parser, and extending it was not in the scope of our project. Our parser still has some conflicts; some less essential constructs have been left out in an attempt to try to reduce the conflicts while they are not solved. Building a parser for a complex language like *Circus* is a great challenge. As a matter of fact, *Circus* is an extension of an already complex language, Z.

It is in our plans is to implement a new *Circus* parser by extending the Z parser from CZT. Doing so, we can take advantage of the completeness and robustness of the Z parser of CZT. For that, it is necessary not only to extend the grammar rules for *Circus* but also to implement the lexical scanners necessary within the framework. As we are already using the CZT abstract syntax tree for *Circus*, the migration to the new parser will not imply in great modifications in the code of *JCircus*.

In the area of formal verification, we plan prove the soundness of the refinement laws that we have proposed. We also want to formally verify the version of the multi-synchronisation strategy that is currently implemented, which allows more than one multi-synchronisation to take part in an external choice.

JCircus is one in a set of tools that the *Circus* team is currently working on. The *Circus* type checker has already been integrated to *JCircus*. We have a prototypal model checker and a theorem prover under development; a refinement editor is also in the plans. The ultimate goal is to have an integrated environment for supporting the formal development of concurrent reactive systems using *Circus*. All these tools working together will assist the user in all phases of development, making the process much easier and trustable. The work we reported here was an important step towards this major goal.

Appendix A

Translation rules

Process declaration

Rule A.1 Normal process declaration

```
$\llbracket \_ \rrbracket^{ProcDecls} : \mathsf{Program} \mathrel{+\!\!\!\!\rightarrow} \mathsf{N} \mathrel{+\!\!\!\!\rightarrow} \mathsf{JCode}$
$\llbracket \epsilon \rrbracket^{ProcDecls}\ proj = \epsilon$
$\llbracket \mathbf{process}\ P \mathrel{\widehat{=}} ProcDef\ \ ProcDecls \rrbracket^{ProcDecl}\ proj =$
    package $proj$.processes;
    import java.util.*;
    import jcsp.lang.*;
    import $proj$.axiomaticDefinitions.*;
    import $proj$.typing.*;
    public class P implements CSProcess { $\llbracket ProcDef \rrbracket^{ProcDef}$ P }
    $\llbracket ProcDecls \rrbracket^{ProcDecls}\ proj$
```

Rule A.2 Generic process declaration

```
$\llbracket \_ \rrbracket^{ProcDecls} : \mathsf{Program} \mathrel{+\!\!\!\!\rightarrow} \mathsf{N} \mathrel{+\!\!\!\!\rightarrow} \mathsf{JCode}$
$\llbracket \mathbf{process}\ P\ [T_0, ..., T_n] \mathrel{\widehat{=}} ProcDef\ \ ProcDecls \rrbracket^{ProcDecls}\ proj =$
    package $proj$.processes;
    import java.util.*;
    import jcsp.lang.*;
    import $proj$.axiomaticDefinitions.*;
    import $proj$.typing.*;
    public class P implements CSProcess {
        $(\llbracket t_0 : \mathbb{N};\ \ldots;\ t_n : \mathbb{N};\ Decl \bullet Proc \rrbracket^{ProcDef}\ P\ \epsilon)$
            [Type,...,Type/$JType\ T_0$,...,$JType\ T_n$]
            [t_0.intValue(),...,t_n.intValue()/
            Type.($Capitals(JType(T_0))$),..., Type.($Capitals(JType(T_n))$)]
    }
    $\llbracket ProcDecls \rrbracket^{ProcDecls}\ proj$
```

Process definition

Rule A.3 Non-parametrised process definition

$[\![\_]\!]^{ProcDef} : \mathsf{ProcDef} \nrightarrow \mathsf{N} \nrightarrow \mathsf{JCode}$
$[\![Proc]\!]^{ProcDef}\ P\ =$

```
    $ChannelDecl\ VisChanEnv\ ChanTypeEnv\ SyncCommEnv$
    $ChannelDecl\ HidChanEnv\ ChanTypeEnv\ SyncCommEnv$
    public $P$(($VisibleCArgs\ VisChanEnv\ ChanTypeEnv$
                              $SyncCommEnv$)){
        $MultiAssign\ (ChannelDecl\ VisChanEnv\ ChanTypeEnv$
                                   $SyncCommEnv)$
                    $(VisibleCArgs\ VisChanEnv\ ChanTypeEnv$
                                   $SyncCommEnv)$
        $HiddenCCreation\ HidChanEnv\ ChanTypeEnv$
                        $SyncCommEnv\ TypesEnv\ MultiSyncEnv\ ReadWriteEnv\ P$
    }

    public void run(){ $ProcessCall\ Proc\ HidChanEnv\ MultiSyncEnv$ }
```

$ChannelDecl : \mathrm{seq}\,\mathsf{N} \nrightarrow (\mathsf{N} \nrightarrow (\mathrm{seq}\,\mathsf{Expr} \times \mathrm{seq}\,\mathsf{Expr})) \nrightarrow (\mathsf{N} \rightarrow SC) \nrightarrow \mathsf{JCode}$
$ChannelDecl\ \langle\rangle \delta\ \zeta = \epsilon$
$ChannelDecl\ (\langle c\rangle \frown cs)\ \delta\ \zeta =$

```
    private GeneralChannel
        $(ArrayDim\ (fst\,(\delta\ c))\ (snd\,(\delta\ c))\ (\zeta\ c)\ 0)$ c;
    $ChannelDecl\ cs\ \delta\ \zeta$
```

$VisibleCArgs : \mathrm{seq}\,\mathsf{N} \nrightarrow (\mathsf{N} \nrightarrow (\mathrm{seq}\,\mathsf{Expr} \times \mathrm{seq}\,\mathsf{Expr})) \nrightarrow (\mathsf{N} \rightarrow SC) \nrightarrow \mathsf{JCode}$
$VisibleCArgs\ \langle\rangle \delta\ \zeta = \epsilon$
$VisibleCArgs\ (\langle c\rangle \frown cs)\ \delta\ \zeta =$

```
    GeneralChannel $(ArrayDim\ (fst\,(\delta\ c))\ (snd\,(\delta\ c))\ (\zeta\ c)\ 0)$ newc,
    $VisibleCArgs\ cs\ \delta\ \zeta$
```

$MultiAssign : \mathsf{JCode} \nrightarrow \mathsf{JCode} \nrightarrow \mathsf{JCode}$

```
$MultiAssign$ (private GeneralChannel v_1 ;...; private GeneralChannel v_n;)
          (GeneralChannel newv_1 ;..., GeneralChannel newv_n;) =
    this.v_1 = newv_1; ...; this.v_n = newv_n;
```

```
$HiddenCCreation : \mathrm{seq}\,\mathsf{N} \nrightarrow (\mathsf{N} \nrightarrow (\mathrm{seq}\,\mathsf{Expr} \times \mathrm{seq}\,\mathsf{Expr})) \nrightarrow (\mathsf{N} \rightarrow SC) \nrightarrow$
                    $\mathrm{seq}\,\mathsf{Expr} \nrightarrow (\mathsf{N} \nrightarrow (\mathsf{N} \nrightarrow \mathbb{N})) \nrightarrow$
                    $(\mathsf{N} \nrightarrow (\mathsf{N} \nrightarrow ChanUse)) \nrightarrow \mathsf{N} \nrightarrow \mathsf{JCode}$
$HiddenCCreation\ \varnothing\ \delta\ \zeta\ types\ \omega\ \alpha\ P = \epsilon$
$HiddenCCreation\ (\langle c \rangle \frown cs)\ \delta\ \zeta\ types\ \omega\ \alpha\ P =$
    **let** $brackets = (ArrayDim\ (fst\,(\delta\ c))\ (snd\,(\delta\ c))\ (\zeta\ c)\ 0)$ **in**
        **if** $(brackets = \epsilon)$ **then**
            **if** $(c \notin \mathrm{dom}\,\omega)$ **then**
                $InitChanInfoSS\ (\alpha\ c)\ c$
                this.c = new GeneralChannel(
                    new Any2OneChannel(),
                    chanInfo_c,
                    P
                );
            **else**
                $InitChanInfoMS\ (\omega\ c)\ c$
                this.c = new GeneralChannel(
                    new Any2OneChannel(),
                    Any2OneChannel.create($\#\omega$),
                    chanInfo_c,
                    P
                );
        **else**
            **if** $(c \notin \mathrm{dom}\,\omega)$ **then**
                $InitChanInfoSS\ (\alpha\ c)\ c$
                this.c = $(InstArray\ (fst\,(\delta\ c))\ (snd(\delta\ c))\ sc$
                            $\tau$ GeneralChannel $\epsilon\ BCGenChanSimple)$
            **else**
                $InitChanInfoMS\ (\omega\ c)\ c$
                Any2OneChannel from_c $brackets$[] =
                    $(InstArray\ (fst\,(\delta\ c))\ (snd(\delta\ c))\ sc\ \tau$ Any2OneChannel $\epsilon\ BCChanFrom)$
                Any2OneChannel to_c $brackets$ =
                    $(InstArray\ (fst\,(\delta\ c))\ (snd(\delta\ c))\ sc\ \tau$ Any2OneChannel $\epsilon\ BCChanTo)$
                this.c =
                    $(InstArray\ (fst\,(\delta\ c))\ (snd(\delta\ c))\ sc\ \tau$ GeneralChannel $\epsilon\ BCGenChanMult)$
        $HiddenCCreation\ cs\ \delta\ \zeta\ types$
```

$Index ::= Type\langle\langle \mathsf{Expr} \rangle\rangle \mid Int\langle\langle \mathbb{N} \rangle\rangle$

$BCGenChanSimple : \mathrm{seq}\ Index \nrightarrow \mathsf{JCode}$
$BCGenChanSimple\ \beta =$ `new GeneralChannel(new Any2OneChannel(), chanInfo_c,` $procName$`)`

$BCChanTo : \mathrm{seq}\ Index \nrightarrow \mathsf{JCode}$
$BCChanTo\ \beta =$ `new Any2OneChannel()`

$BCChanFrom : \mathrm{seq}\ Index \nrightarrow \mathsf{JCode}$
$BCChanFrom\ \beta =$ `Any2OneChannel.create(`$\#\omega$`)`

```
$BCGenChanMult$ : seq $Index \nrightarrow$ JCode
$BCGenChanMult\ \beta =$ new GeneralChannel(to_c($Indexes\ \beta$)
     from_c($Indexes\ \beta$), chanInfo_c, $procName$)
```

```
$BCMSCtrl$ : seq $Index \nrightarrow$ JCode
$BCMSCtrl\ \beta =$ new MultiSyncControl(c($Indexes\ \beta$).getFromController(),
    c($Indexes\ \beta$).getToController())
```

```
$Indexes$ : seq $Index \nrightarrow$ JCode
$Indexes\ \langle Type\langle\langle T\rangle\rangle\rangle \frown ts =$ [Type.($CType\ T$)] $Indexes\ ts$
$Indexes\ \langle Int\langle\langle n\rangle\rangle\rangle \frown ns =$ [$n$] $Indexes\ ns$
```

```
$InstArray$ : seq Expr $\nrightarrow$ seq Expr $\nrightarrow$ $SC$ $\nrightarrow$ seq Expr $\nrightarrow$
                JCode $\nrightarrow$ seq $Index \nrightarrow$ ($seq\ Index \nrightarrow$ JCode) $\nrightarrow$ JCode
$InstArray\ genTypes\ types\ sc\ \tau\ nmComp\ \beta\ func =$
    let $dim = (ArrayDim\ genTypes\ types\ sc)$ in
        if ($\#genTypes > 0$) then
            new $nmComp\ dim${ $GenericInst\ genTypes\ types\ sc\ \tau\ \tau\ nmComp\ \beta\ func$ }
        else $InstArraySync\ types\ sc\ nmComp\ \beta\ func$
```

```
$GenericInst$ : seq Expr $\nrightarrow$ seq Expr $\nrightarrow$ $SC$ $\nrightarrow$ seq Expr $\nrightarrow$ seq Expr $\nrightarrow$
                JCode $\nrightarrow$ seq $Index \nrightarrow$ ($seq\ Index \nrightarrow$ JCode) $\nrightarrow$ JCode
$GenericInst\ genTypes\ types\ sc\ \tau\ \langle T\rangle\ nmComp\ \beta\ func =$
    $InstArray\ (tail\ genTypes)\ (replace(head\ genTypes, T, types))\ sc$
                                      $\tau\ nmComp\ \beta \frown \langle\ Type\langle\langle T\rangle\rangle\rangle\ func$
$GenericInst\ genTypes\ types\ sc\ \tau\ \langle T\rangle \frown ts\ nmComp\ \beta\ func =$
    $InstArray\ (tail\ genTypes)\ (replace(head\ genTypes, T, types))\ sc$
                                      $\tau\ nmComp\ \beta \frown \langle\ Type\langle\langle T\rangle\rangle\rangle\ func$,
    $GenericInst\ genTypes\ types\ sc\ \tau\ ts\ nmComp\ \beta\ func$
```

```
$InstArraySync$ : seq Expr $\nrightarrow$ $SC$ $\nrightarrow$ JCode $\nrightarrow$ seq $Index \nrightarrow$ ($seq\ Index \nrightarrow$ JCode) $\nrightarrow$ JCode
$InstArraySync\ types\ sc\ nmComp\ \beta\ func =$
    let $type =$ if $\#types > 0$ then $(head\ types)$ else $null$,
        $dim = (ArrayDimSync\ types\ sc\ 0)$ in
        if ($\#types = 1\ and\ sc = C$) $or\ \#types = 0$ then
            ($func\ \beta$)
        else new $nmComp\ dim$
            { $TypeInstSync\ types\ sc\ Max(JType(type))\ nmComp\ \beta\ func$ }
```

```
$TypeInstSync$ : seq Expr $\nrightarrow$ $SC$ $\nrightarrow \mathbb{N} \nrightarrow$ JCode $\nrightarrow$ seq $Index \nrightarrow$ ($seq\ Index \nrightarrow$ JCode) $\nrightarrow$ JCode
$TypeInstSync\ types\ sc\ 1\ nmComp\ \beta\ func =$
    $InstArraySync\ (tail\ types)\ sc\ nmComp\ \beta \frown \langle Int\langle\langle 1\rangle\rangle\rangle\ func$
$TypeInstSync\ types\ sc\ n\ nmComp\ \beta\ func =$
    $InstArraySync\ (tail\ types)\ sc\ nmComp\ \beta \frown \langle Int\langle\langle n\rangle\rangle\rangle\ func$,
    $TypeInstSync\ types\ sc\ (n-1)\ nmComp\ \beta\ func$
```

```
$ArrayDim : \operatorname{seq} \mathsf{Expr} \rightarrow\!\!\!\!\!+ \operatorname{seq} \mathsf{Expr} \rightarrow\!\!\!\!\!+ SC \rightarrow\!\!\!\!\!+ \mathsf{JCode}$
$ArrayDim\ genTypes\ types\ sc =$
    $\mathbf{let}\ \ dim = \#genTypes + (ArrayDimSync\ types\ sc) =$
    $\mathbf{in}$ [] $^{dim}$
```

```
$ArrayDimSync : \operatorname{seq} \mathsf{Expr} \rightarrow\!\!\!\!\!+ SC \rightarrow\!\!\!\!\!+ \mathsf{JCode}$
$ArrayDimSync\ types\ sc =$
    $\mathbf{let}\ dim = \mathbf{if}\ types = \langle Sync \rangle\ \mathbf{then}\ 0$
                  $\mathbf{else\ if}\ sc = C\ \mathbf{then}\ \#types - 1$
                  $\mathbf{else}\ \#types$
    $\mathbf{in}$ [] $^{dim}$
```

```
$InitChanInfoMS : (\mathsf{N} \rightarrow\!\!\!\!\!+ \mathbb{N}) \rightarrow\!\!\!\!\!+ \mathsf{N} \rightarrow\!\!\!\!\!+ \mathsf{JCode}$
$InitChanInfoMS\ \varnothing\ c = \epsilon$
$InitChanInfoMS\ (procName \mapsto n) \cup \alpha\ c =$
    ChanInfo chanInfo_c = new ChanInfo();
    chanInfo_c.put($procName$, new Integer($n$));
    $InitChanInfoMS\ \alpha\ c$
```

```
$InitChanInfoSS : (\mathsf{N} \rightarrow\!\!\!\!\!+ (\mathsf{N} \rightarrow\!\!\!\!\!+ ChanUse)) \rightarrow\!\!\!\!\!+ \mathsf{N} \rightarrow\!\!\!\!\!+ \mathsf{JCode}$
$InitChanInfoSS\ \varnothing\ c = \epsilon$
$InitChanInfoSS\ (procName \mapsto I) \cup \alpha\ c =$
    ChanInfo chanInfo_c = new ChanInfo();
    chanInfo_c.put($procName$, new Integer(1));
    $InitChanInfoSS\ \alpha\ c$
$InitChanInfoSS\ (procName \mapsto O) \cup \alpha\ c =$
    ChanInfo chanInfo_c = new ChanInfo();
    chanInfo_c.put($procName$, new Integer(0));
    $InitChanInfoSS\ \alpha\ c$
```

```
$ProcessCall : \mathsf{ProcDef} \rightarrow\!\!\!\!\!+ \operatorname{seq} \mathsf{N} \rightarrow\!\!\!\!\!+ (\mathsf{N} \rightarrow\!\!\!\!\!+ (\mathsf{N} \rightarrow\!\!\!\!\!+ \mathbb{N})) \rightarrow\!\!\!\!\!+ JCode$
$ProcessCall\ Proc\ \iota\ \omega =$
    $\mathbf{if}\ (\operatorname{ran} \iota \cap \operatorname{dom} \omega) \neq \varnothing\ \mathbf{then}$
        $\mathbf{let}\ (\operatorname{ran} \iota \cap \operatorname{dom} \omega) = \{c_1, ..., c_n\}\ \mathbf{in}$
            Any2OneChannel endManager = new Any2OneChannel();
            $InitChanInfoMS\ (fst\,\omega)\ P$
            $[\![ProcessManagerMultiSync(Proc)\ || \left( ControllersManager \left( \begin{array}{l} MultiSyncControl(from\_c_1, to\_c_1) \\ ||\ ... \\ ||\ MultiSyncControl(from\_c_n, to\_c_n) \end{array} \right) \right)]\!]^{Proc}$
    $\mathbf{else}\ [\![\,Proc]\!]^{Proc}$
```

```
$[\![MultiSyncControl(from_c, to_c)]\!]^{Proc} =$
    new MultiSyncControl(from_c, to_c).run();
```

```
$[\![ProcessManagerMultiSync(Proc)]\!]^{Proc}$ =
    (new ProcessManagerMultiSync(
        endManager,
        new CSProcess() { public void run() { $[\![ Proc ]\!]^{Proc}$ }}
    )).run();
```

```
$[\![ControllersManager(Proc)]\!]^{Proc}$ =
    (new ControllersManager(
        endManager,
        new CSProcess() { public void run() { $[\![ Proc ]\!]^{Proc}$ }}
    )).run();
```

Rule A.4 Parametrised process definition

```
$[\![Decl \bullet Proc]\!]^{ProcDef}\ P$ =
    $ParamsDecl\ Decl$
    $ChannelDecl\ VisChanEnv\ ChanTypeEnv\ SyncCommEnv$
    $ChannelDecl\ HidChanEnv\ ChanTypeEnv\ SyncCommEnv$
    public $P$($ParamsArgs\ Decl$,
              ($VisibleCArgs\ VisChanEnv\ ChanTypeEnv$
                                  $SyncCommEnv$)){
        $MultiAssign\ (ParamsDecl\ Decl)\ (ParamsArgs\ Decl)$
        $MultiAssign\ (ChannelDecl\ VisChanEnv\ ChanTypeEnv$
                                  $SyncCommEnv)$
                   $(VisibleCArgs\ VisChanEnv\ ChanTypeEnv$
                                  $SyncCommEnv)$
        $HiddenCCreation\ HidChanEnv\ ChanTypeEnv$
                        $SyncCommEnv\ TypesEnv$
    }
    public void run(){ $ProcessCall\ Proc\ HidChanEnv\ MultiSyncEnv$ }
```

$ParamsDecl : \mathsf{Decl} \nrightarrow \mathsf{JCode}$
$ParamsDecl\ x_1 : T_1;\ \ldots;\ x_n : T_n =$

```
    private ($JType\ T_1$) x_1;...; private ($JType\ T_n$) x_n;
```

$ParamsArgs : \mathsf{Decl} \nrightarrow \mathsf{JCode}$
$ParamsArgs\ x_1 : T_1;\ \ldots;\ x_n : T_n =$

```
    ($JType\ T_1$) newx_1,..., ($JType\ T_n$) newx_n
```

Rule A.5 Indexed process definition

$$[\![x_1 : T_1;\ \ldots;\ x_n : T_n \odot Proc\ ]\!]^{ProcDef}\ P =$$
$$[\![(x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Proc)[c : used(Proc) \bullet c := c\_x\_1 \ldots x\_n]\ ]\!]^{ProcDef}\ P$$

Processes

Rule A.6 Basic process

$[\![\_]\!]^{Proc}$: $\mathsf{Process} \nrightarrow \mathsf{JCode}$
$[\![\mathbf{begin}\ PPars_1\ \mathbf{state}\ PSt\ PPars_2 \bullet Main]\!]^{Proc} =$

```
(new CSProcess(){
    $StateDecl\ PSt$
    $[\![PPars_1\ PPars_2]\!]^{PPars}$
    public void run() { $[\![Main]\!]^{Action}$ }
}).run();
```

$StateDecl : \mathsf{SchemaExp} \nrightarrow \mathsf{JCode}$
$StateDecl\ [\, x_1 : T_1;\ \ldots;\ x_n : T_n \mid inv\,] =$
`private` $(JType\ T_1)$ `x_1;`$\ldots$`; private` $(JType\ T_n)$ `x_n;`

Rule A.7 Process call

$[\![N]\!]^{Proc} =$ `(new N(`$ExtractChans\ VisChanEnv\ N$`)).run();`

Rule A.8 Parametrised process call

$[\![N(e_1, \ldots, e_n)]\!]^{Proc} =$

```
(new N(($JExp\ e_1$),...,($JExp\ e_n$),
        ($ExtractChans\ VisChanEnv\ N$))).run();
```

$ExtractChans : \mathrm{seq}\,\mathsf{N} \nrightarrow \mathsf{N} \nrightarrow \mathsf{JCode}$
$ExtractChans\ \langle c\rangle \frown \langle\rangle\ N =$ `new GeneralChannel(c, N)`
$ExtractChans\ \langle c\rangle \frown cs\ N =$ `c,(`$ExtractChans\ cs\ N$`)`, $cs \neq \langle\rangle$

Rule A.9 Implicit parametrised process call

$[\![(Decl \bullet Proc)(e_1, \ldots, e_n)]\!]^{Proc} =$

```
(new CSProcess(){
    public void run() {
        $DeclareProcessClass\ Decl\ Proc\ index$
        I_$index$ i_$index$_$index$ =
            new I_$index$(($JExp\ e_1$),...,($JExp\ e_n$),
                    ($JList\ (ListFirst\ LocalVarEnv)$));
        i_$index$_$index$.run();
    }
}).run();
```

$DeclareProcessClass : \mathsf{Decl} \nrightarrow \mathsf{Proc} \nrightarrow \mathbb{N} \nrightarrow \mathsf{JCode}$
$DeclareProcessClass\ Decl\ Proc\ index =$

```
class I_$index$ implements CSProcess {
    $ParamsDecl\ Decl$
    public I_$index$($ParamsArgs\ Decl$){
        $MultiAssign\ (ParamsDecl\ Decl)\ (ParamsArgs\ Decl)$
    }
    public void run (){
        $[\![Proc]\!]^{Proc}$
    }
}
```

Rule A.10 Indexed call

$$[\![\, N\lfloor v_1, \ldots, v_n\rfloor \,]\!]^{Proc} = [\![\, N(v_1, \ldots, v_n) \,]\!]^{Proc}$$

Rule A.11 Implicit indexed call

$[\![\_]\!]^{Proc} : \mathsf{Proc} \nrightarrow \mathsf{JCode}$

$$[\![\, (x_1 : T_1;\ \ldots;\ x_n : T_n \odot Proc)\lfloor v_1, \ldots, v_n\rfloor \,]\!]^{Proc} =$$
$$[\![\, ((x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Proc)[c : used(Proc) \bullet c := c\_x\_1 \ldots x\_n]) \\ (v_1, \ldots, v_n) \,]\!]^{Proc}$$

Rule A.12 Generic instantiation

$[\![\, N[T_0, \ldots, T_n] \,]\!]^{Proc} =$

```
(new N(new Integer(Type.($Capitals\ JType(T_0)$)),...,
        new Integer(Type.($Capitals\ JType(T_n)$)),
        $ExtractChans\ VisChanEnv$)).run();
```

Rule A.13 Generic instantiation and parametrised call

$[\![\, N[T_0, \ldots, T_n](e_1, \ldots, e_n) \,]\!]^{Proc} =$

```
(new N(new Integer(Type.($Capitals\ JType(T_0)$)),...,
        new Integer(Type.($Capitals\ JType(T_n)$)),
        ($JExp\ e_1$),...,($JExp\ e_n$),
        $ExtractChans\ VisChanEnv$)).run();
```

Rule A.14 Channel renaming

$[\![Proc[x_1, \ldots, x_n := y_1, \ldots, y_n]]\!]^{Proc} =$
$[\![Proc\,]\!]^{Proc}$ `[y_1,...,y_n/x_1,...,x_n]`

Rule A.15 Hiding

$$[\![Proc \setminus \{\!|\ cs\ |\!\}]\!]^{Proc} = [\![Proc]\!]^{Proc}$$

Rule A.16 Sequential composition

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

```
(new CSProcess(){
    public void run() {
        $[\![Proc_1\,]\!]^{Proc}$ ; ... ; $[\![\,Proc_n]\!]^{Proc}$
    }
}).run();
```

Rule A.17 Internal choice

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

```
(new CSProcess(){
    public void run() {
        int choosen = RandomGenerator.generateNumber(1,n);
        switch(choosen) {
            case 1:
                { $[\![\,Proc_1\,]\!]^{Proc}$ }
                break;
            ...
            case n:
                { $[\![\,Proc_n\,]\!]^{Proc}$ }
                break;
        }
    }
}).run();
```

Rule A.18 Process parallelism

$[\![Proc_1 \;[\![\; CSExp \;]\!]\; Proc_2]\!]^{Proc} =$

```
new Parallel (
        new CSProcess[] {
            new CSProcess() { public void run() { $[\![\,Proc_1\,]\!]^{Proc}$ } },
            new CSProcess() { public void run() { $[\![\,Proc_2\,]\!]^{Proc}$ } }
        }
```

Rule A.19 Process interleaving

$[\![Proc_1 \;|||\; Proc_2]\!]^{Proc} = [\![Proc_1 \;[\![\; \{\!|\,|\!\} \;]\!]\; Proc_2]\!]^{Proc}$

Rule A.20 Iterated sequential composition

```
$[\![ \fatsemi\, x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Proc ]\!]^{Proc} =$
    (new CSProcess(){
        public void run() {
            $InstProcesses$ procVec_$index$ $(x_1 : T_1;\ \ldots;\ x_n : T_n)$ $Proc$ $index$
            for (int i = 0; i<procVec_$index$.size(); i++){
                ((CSProcess)procVec_$index$.get(i)).run();
            }
        }
    }).run();
```

```
$InstProcesses : \mathsf{N} \leftrightarrow \mathsf{Decl} \leftrightarrow \mathsf{Proc} \leftrightarrow \mathbb{N} \leftrightarrow \mathsf{JCode}$
$InstProcesses$ $procVecName$ $(x_1 : T_1;\ \ldots;\ x_n : T_n)$ $Proc$ $index =$
    Vector $procVecName$ = new Vector();
    $DeclareProcessClass$ $(x_1 : T_1;\ \ldots;\ x_n : T_n)$ $Proc$ $index$
    for (($JType$ $T_1$) x_1 = ($Min$ $T_1$);
          x_1.compareTo(($Max$ $T_1$))<=0; ($Inc$ $x_1 : T_1$)){
        ...
        for (($JType$ $T_n$) x_n = ($Min$ $T_n$);
              x_n.compareTo(($Max$ $T_n$))<=0; ($Inc$ $x_n : T_n$)){
            $procVecName$.add(new I_$index$(x_1,...,x_n);
        }
        ...
    }
```

Rule A.21 Iterated internal choice

```
$[\![ \sqcap\, x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Proc ]\!]^{Proc} =$
    (new CSProcess(){
        public void run() {
            $ChooseIndexVars$ $(x_1 : T_1;\ \ldots;\ x_n : T_n)$
            $DeclareProcessClass$ $(x_1 : T_1;\ \ldots;\ x_n : T_n)$ $Proc$ $index$
            (new I_$index$(x_1,...,x_n)).run();
        }
    }).run();
```

```
$ChooseIndexVars : \mathsf{Decl} \leftrightarrow \mathsf{JCode}$
$ChooseIndexVars\ \epsilon = \epsilon$
$ChooseIndexVars$ $(x : T;\ Decls) =$
    ($JType$ $T$) x =
        new ($JType$ $T$)(RandomGenerator.generateNumber(($Min$ $T$),($Max$ $T$)));
    $ChooseIndexVars$ $Decls$
```

Rule A.22 Iterated process parallelism

$[\![\, [\![\, \mathit{CSExp} \,]\!]\; x_1 : T_1;\ \ldots;\ x_n : T_n \bullet \mathit{Proc}]\!]^{\mathit{Proc}} =$

```
(new CSProcess(){
    public void run() {
        InstProcesses procVec_index (x1 : T1; ...; xn : Tn) Proc index
        CSProcess[] processes_index =
            new CSProcess[procVec_index.size()];
        for (int i = 0; i < procVec_index.size(); i++){
            processes_index[i] =
                (CSProcess)procVec_index.get(i);
        }
        (new Parallel(processes_index)).run();
    }
}).run();
```

Rule A.23 Iterated interleaving

$$[\![\, |\!|\!|\; x_1 : T_1;\ \ldots;\ x_n : T_n \bullet \mathit{Proc}]\!]^{\mathit{Proc}} = [\![\, [\![\, \{\!|\,|\!\}\,]\!]\; x_1 : T_1;\ \ldots;\ x_n : T_n \bullet \mathit{Proc}]\!]^{\mathit{Proc}}$$

Process paragraphs

Rule A.24 Axiomatic definition

$[\![\_]\!]^{PParags} : \mathsf{PParagraph}^* \nrightarrow \mathsf{JCode}$

$[\![\epsilon]\!]^{PParags} = \epsilon$

$[\![v : T \mid v = e_1\ PPars]\!]^{PParags} =$

```
    private (JType T) v() { return (JExp e1); }
    [[PParags]]^PParags
```

Rule A.25 Non-parametrised action definition

$[\![N \mathrel{\widehat{=}} \mathit{Action}\ \mathit{PParags}]\!]^{PParags} =$

```
    private void  N(){  [[Action]]^Action  }
    [[PParags]]^PParags
```

Rule A.26 Parametrised action definition

$[\![N \mathrel{\widehat{=}} (\mathit{Decl} \bullet \mathit{Action})\ \mathit{PParags}]\!]^{PParags} =$

```
    private void N(ParamsArgs Decl){  [[Action]]^Action  }
    [[PParags]]^PParags
```

Rule A.27 Recursive parametrised action definition

$[\![N \mathrel{\widehat{=}}\ \mu X \bullet (\mathit{Decl} \bullet \mathit{Action}(X(e_0, \ldots, e_n)))\mathit{PParags}]\!]^{PParags} =$

$\quad [\![N \mathrel{\widehat{=}}\ (\mathit{Decl} \bullet \mathit{Action}(N(e_0, \ldots, e_n)))]\!]^{PParags}$

$\quad [\![\mathit{PParags}]\!]^{PParags}$

Actions

CSP Actions

Rule A.28 Skip

$[\![\_]\!]^{Action} : \mathsf{Action} \nrightarrow \mathsf{JCode}$
$[\![Skip]\!]^{Action} =$ `(new Skip()).run();`

Rule A.29 Stop

$[\![Stop]\!]^{Action} =$ `(new Stop()).run();`

Rule A.30 Chaos

$[\![Chaos]\!]^{Action} =$ `while(true){};`

Rule A.31 Prefixing action

$[\![c \rightarrow Action]\!]^{Action} =$ `c.synchronise();`

Rule A.32 Prefixing action – input

$[\![c?x \rightarrow Action]\!]^{Action} =$
  let $commType = JType(last\ (snd\ (ChanTypeEnv\ c)))$ ***in***
    `{` $commType$ `x = (`$commType$`)c.read();`
      $[\![Action]\!]^{Action}$ `}`

Rule A.33 Prefixing action – output

$[\![c!e \rightarrow Action]\!]^{Action} =$ `c.write(`$JExp\ e$`);` $[\![Action]\!]^{Action}$

Rule A.34 Prefixing action – generic

$[\![c\ [T_0, \ldots, T_n].e_0\ \ldots\ .e_m \rightarrow Action]\!]^{Action} =$
  `c[Type.(`$CJType\ T_0$`)]` ... `[Type.(`$CJType\ T_n$`)]`
   `[(`$JExp\ e_0$`).getValue()]` ... `[(`$JExp\ e_n$`).getValue()].synchronise();`
  $[\![Action]\!]^{Action}$

Rule A.35 Prefixing action – generic and input

$\llbracket c\ [T_0, \ldots, T_n].e_0 \ldots .e_m?x \rightarrow Action \rrbracket^{Action} =$

```
    let commType = JType(last (snd (δ c))) in
        { commType x =
              (commType)c[Type.(CJType T0)]...[Type.(CJType Tn)]
                    [(JExp e0).getValue()]...[(JExp en).getValue()].read();
          ⟦Action⟧^Action }
```

Rule A.36 Prefixing action – generic and output

$\llbracket c\ [T_0, \ldots, T_n].e_0 \ldots .e_m!x \rightarrow Action \rrbracket^{Action} =$

```
    c[Type.(CJType T0)]...[Type.(CJType Tn)]
     [(JExp e0).getValue()]...[(JExp en).getValue()].write(JExp x);
    ⟦Action⟧^Action
```

Rule A.37 Guarded action

```
⟦Pred & Action⟧^Action = if(JExp(Pred)) {
                              ⟦Action⟧^Action
                         } else {
                              (new Stop()).run();
                         }
```

Rule A.38 Sequential composition

$$\llbracket Action_1;\ Action_2 \rrbracket^{Action} = \llbracket Action_1 \rrbracket^{Action}\ ;\ \llbracket Action_2 \rrbracket^{Action}$$

Rule A.39 External choice

$\llbracket A_1 \Box \ldots \Box A_m \rrbracket^{Action} =$

```
    DeclConst A1 1
    ...
    DeclConst Am m
    RunClient ω ind (InitChanName A1) ⁀ ... ⁀ (InitChanName Am)
    switch(client_ind.getChoosen()) {
        Case A1
        ...
        Case Am
    }
```

$DeclConst$: Action $\nrightarrow \mathbb{N} \nrightarrow$ JCode
$DeclConst\ A\ n =$ `final int` $(ConstChan\ A)$ `=` n`;`

$Case$: Action $\nrightarrow$ JCode
$Case\ A =$

```
    case (ConstChan A):
        { ⟦A⟧^Action }
        break;
```

$ConstChan : \mathsf{Action} \nrightarrow \mathsf{JCode}$
$ConstChan\ (c \rightarrow Action) = \texttt{CONST\_}(Capitals\ c)$
$ConstChan\ (c[T_0, \ldots, T_n].x_0 \ldots .x_m) =$
    $\texttt{CONST\_}(Capitals\ c)\texttt{\_}Capitals(JType(T_0))\texttt{\_} \ldots \texttt{\_}Capitals(JType(T_n))\texttt{\_X\_0\_} \ldots \texttt{X\_m}$
$ConstChan\ (g\ \&\ Action) = ConstChan\ Action$

$InitChanName : \mathsf{Action} \nrightarrow \mathsf{N}$
$InitChanName\ (c \rightarrow Action) = c$
$InitChanName\ (c[T_0, \ldots, T_n].x_0 \ldots .x_m) = c$
$InitChanName\ (g\ \&\ Action) = InitChanName\ Action$

$RunClient : (\mathsf{N} \nrightarrow (\mathsf{N} \nrightarrow \mathbb{N})) \nrightarrow \mathbb{N} \nrightarrow \operatorname{seq} \mathsf{N} \nrightarrow \mathsf{JCode}$
$RunClient\, \omega\ ind\ c_1 \frown \ldots \frown c_m =$

```
    Vector seqOfSync_ind = new Vector();
    Vector seqOfNotSync_ind = new Vector();
    boolean g[] = { Guard A_1,..., Guard A_m };
    if (c_1.isMultiSync()) {
        Object[] sync = new Object[] {
            c_1.getFromControlerId(),
            c_1.getChannel(),
            new Integer(c_1.getProcessId()),
            new Integer(0)
        };
        seqOfSync_ind.addElement(sync);
    } else {
        seqOfNotSync_ind.addElement(c_1.getChannel());
    }
    ...
    if (c_m.isMultiSync()) {
        Object[] sync = new Object[] {
            c_m.getFromControlerId(),
            c_m.getChannel(),
            new Integer(c_m.getProcessId()),
            new Integer(0)
        };
        seqOfSync_ind.addElement(sync);
    } else {
        seqOfNotSync_ind.addElement(c_m.getChannel());
    }
    MultiSyncClient client_ind =
        new MultiSyncClient(seqOfSync_ind,seqOfNotSync_ind, null, g);
    client_ind.run();
```

$Guard : \mathsf{Action} \nrightarrow \mathsf{JCode}$
$Guard\ (comm \rightarrow Action) = \texttt{true}$
$Guard\ (g\ \&\ Action) = (JExp\ g)\ \texttt{\&\&}\ Guard\ Action$

Rule A.40 Internal choice

```
$[\![Action_1 \sqcap \ldots \sqcap Action_n]\!]^{Action}$ =
    int choosen = RandomGenerator.generateNumber(1,n);
    switch(choosen) {
        case 1: { $[\![Action_1]\!]^{Action}$ } break;
        ...
        case n: { $[\![Action_n]\!]^{Action}$ } break;
    }
```

Rule A.41 Action parallelism

```
$[\![Action_1 \; [\![ NSExp_1 \mid CSExp \mid NSExp_2 ]\!] \; Action_2]\!]^{Action}$ =
    let LName = ParallelLeftBranch_index,
        RName = ParallelRightBranch_index in
            DeclareClassesActionCall ActionEnv BasicProcEnv
            class LName implements CSProcess {
                InitAuxVars (setFirst StateCompEnv)) index L
                DeclLocalVars LocalVarEnv index L
                public LName((LocalVarsArg LocalVarEnv)) {
                    InitLocalVars LocalVarEnv index L
                }
                public void run () {
                    RenameVars $[\![Action_1]\!]^{Action}$
                                ((SetFirst StateCompEnv) ∪ (SetFirst LocalVarEnv))
                                index L
                }
            }
            CSProcess left_index =
                new LName(JList (ListFirst LocalVarEnv));
            class RName implements CSProcess {
                InitAuxVars ((setFirst StateCompEnv)) index R
                DeclLocalVars LocalVarEnv index R
                public RName((LocalVarsArg LocalVarEnv)) {
                    InitLocalVars LocalVarEnv index R
                }
                public void run () {
                    RenameVars $[\![Action_2]\!]^{Action}$
                                ((SetFirst StateCompEnv) ∪ (SetFirst LocalVarEnv))
                                index R
                }
            }
            CSProcess right_index =
                new RName(JList (ListFirst LocalVarEnv));
            CSProcess[] processes_index =
                new CSProcess[]{left_index,right_index};
            (new Parallel(processes_index)).run ();
            MergeVars LName $NSExp_1$ index L
            MergeVars RName $NSExp_2$ index R
```

$DeclareClassesActionCall : \text{seq}(\mathsf{N} \times \mathsf{Action}) \pfun \text{seq}(\mathsf{N} \times \mathsf{Expr}) \pfun \mathsf{JCode}$
$DeclareClassesActionCall(N, Action) \frown ss\ env =$

```
    class N_ implements CSProcess {
        $DeclLocalVars\ (setFirst\ env)\ 0\ LDeclareClassesActionCall\ ss\ env$
        public N_($LocalVarsArg\ env$){
            $InitLocalVars\ env\ 0\ L$
        }
        public void run() {
            $[\![Action]\!]^{Action}$
        }
```

$SetFirst : \text{seq}(\mathsf{N} \times \mathsf{Expr}) \pfun \mathbb{P}\,\mathsf{N}$
$SetFirst\ \langle\rangle = \varnothing$
$SetFirst\ \langle x, T\rangle \frown xs = \{x\} \cup (SetFirst\ xs)$

$InitAuxVars : \mathbb{P}\,\mathsf{N} \pfun \mathbb{N} \pfun LeftRight \pfun \mathsf{JCode}$
$InitAuxVars\ \varnothing\ index\ S = \epsilon$
$InitAuxVars\ (\{x\} \cup xs)\ index\ L =$

```
    public ($JType\ (CType\ x)$) aux_left_x_$index$ = x;
    $InitAuxVars\ xs\ index\ L$
```

$InitAuxVars\ (\{x\} \cup xs)\ index\ R =$

```
    public ($JType\ (CType\ x)$) aux_right_x_$index$ = x;
    $InitAuxVars\ xs\ index\ R$
```

$DeclLocalVars : \text{seq}(\mathsf{N} \times \mathsf{Expr}) \pfun \mathbb{N} \pfun LeftRight \pfun \mathsf{JCode}$
$DeclLocalVars\ [\,]\ index\ x = \epsilon$
$DeclLocalVars\ ((x, T) : xs)\ index\ L =$

```
    public ($JType\ T$) aux_left_x_$index$; $DeclLocalVars\ xs\ index\ L$
```

$DeclLocalVars\ ((x, T) : xs)\ index\ R =$

```
    public ($JType\ T$) aux_right_x_$index$; $DeclLocalVars\ xs\ index\ R$
```

$LocalVarsArg : \text{seq}(\mathsf{N} \times \mathsf{Expr}) \pfun \mathsf{JCode}$
$LocalVarsArg\ [\,] = \epsilon$
$LocalVarsArg\ (x, T) : [\,] = (JType\ T)$ `x`
$LocalVarsArg\ (x, T) : xs = (JType\ T)$ `x,` $LocalVarsArg\ xs$

$InitLocalVars : \text{seq}(\mathsf{N} \times \mathsf{Expr}) \pfun \mathbb{N} \pfun LeftRight \pfun \mathsf{JCode}$
$InitLocalVars\ [\,]\ index\ x = \epsilon$
$InitLocalVars\ ((x, T) : xs)\ index\ L =$

```
    this.aux_left_x_$index$ = x; $DeclLocalVars\ xs\ index\ L$
```

$InitLocalVars\ ((x, T) : xs)\ index\ R =$

```
    this.aux_right_x_$index$ = x; $DeclLocalVars\ xs\ index\ R$
```

$RenameVars : \mathsf{JCode} \pfun \mathbb{P}\,\mathsf{N} \pfun \mathbb{N} \pfun LeftRight \pfun \mathsf{JCode}$
$RenameVars\ jcode\ \varnothing\ index\ x = \epsilon$
$RenameVars\ jcode\ (\{x\} \cup xs)\ index\ L =$

```
    $RenameVars\ (jcode[$aux_left_x_$index$/x$])\ xs\ index\ L$
```

$RenameVars(jcode, \{x\} \cup xs, index, R) =$

```
    $RenameVars\ (jcode[$aux_right_x_$index$/x$])\ xs\ index\ R$
```

```
$MergeVars : \mathsf{N} \nrightarrow \mathbb{P}\,\mathsf{N} \nrightarrow \mathbb{N} \nrightarrow \{L, R\} \nrightarrow \mathsf{JCode}$
$MergeVars\ name\ \varnothing\ index\ x = \epsilon$
$MergeVars\ LName\ (\{x\} \cup xs)\ index\ L =$
    x = ((LName)processes_$index$[0]).aux_left_x_index;
    $MergeVars\ LName\ xs\ index\ L$
$MergeVars\ RName\ (\{x\} \cup xs)\ index\ R =$
    x = ((RName)processes_$index$[1]).aux_right_x_index;
    $MergeVars\ RName\ xs\ index\ R$
```

Rule A.42 Action interleaving

$$[\![Action_1 \;|\![NSExp_1 \mid NSExp_2]\!|\; Action_2]\!]^{Action} = [\![Action_1 \;[\![\; NSExp_1 \mid \{\![\,]\!\} \mid NSExp_2 \;]\!]\; Action_2]\!]^{Action}$$

Rule A.43 Recursive action

```
$[\![ \mu\, X \bullet Action(X) ]\!]^{Action} =$
    class I_$index$ implements CSProcess {
        $DeclLocalVars\ LocalVarEnv\ index\ L$
        public I_$index$($LocalVarsArg\ LocalVarEnv$) {
            $InitLocalVars\ LocalVarEnv\ index\ L$
         }
        public void run() {
            $RenameVars$
                $[\![ Action((RunRecursion\ index\ \langle\rangle)) ]\!]^{Action}$
                $(SetFirst\ LocalVarEnv)\ index\ L$
        }
    };
    $RunRecursion\ index\ \langle\rangle$
```

```
$RunRecursion : (\mathbb{N} \times \mathrm{seq}\,\mathsf{Expr}) \nrightarrow \mathsf{JCode}$
$RunRecursion\ index\ \langle e_0, \ldots, e_m \rangle =$
    I_$index$ i_$index$_$newIndex$ =
       new I_$index$($JList\ (ListFirst\ LocalVarEnv)$,$JExp(e_0)$, ... ,$JExp(e_n)$);
    i_$index$_$newIndex$.run();
    $MergeLocalVars\ LocalVarEnv\ index\ newIndex\ L$
```

```
$MergeLocalVars : \mathbb{P}\,\mathsf{N} \nrightarrow \mathbb{N} \nrightarrow \mathbb{N} \nrightarrow \{L, R\} \nrightarrow \mathsf{JCode}$
$MergeLocalVars\ \varnothing\ index\ newIndex\ x = \epsilon$
$MergeLocalVars\ (\{x\} \cup xs)\ index\ newIndex\ L =$
    x = i_$index$_$newIndex$.aux_left_x_$index$;
    $MergeLocalVars\ xs\ index\ newIndex\ L$
$MergeLocalVars\ (\{x\} \cup xs)\ index\ newIndex\ R =$
    x = i_$index$_$newIndex$.aux_right_x_$index$;
    $MergeLocalVars\ xs\ index\ newIndex\ R$
```

Rule A.44 Action call

$[\![N]\!]^{Action} = \texttt{N();}$

Rule A.45 Action call (in parallel actions)

```
$[\![N]\!]^{Action} =$
    $\texttt{N\_classn\_}\mathit{index} = \texttt{N\_class(}\mathit{JList}\ (\mathit{ListFirst}\ \mathit{BasicProcEnv})\texttt{);}$
    $\texttt{n\_}\mathit{index}\texttt{.run();}$
    $\mathit{RetrieveVars}\ \mathit{BasicProcEnv}$
```

```
$\mathit{RetrieveVars} : \mathrm{seq}(\mathsf{N} \times \mathsf{Expr} \rightarrowtail \mathsf{N} \rightarrowtail \mathbb{N} \rightarrowtail \mathsf{JCode}$
$\mathit{RetrieveVars}\langle\rangle\ \mathit{nClass}\ \mathit{index}$
$\mathit{RetrieveVars}(v, t) \frown \mathit{ss}\ \mathit{nClass}\ \mathit{index} =$
    $v\ \texttt{=}\ \mathit{nClass\_index}\texttt{.}v\texttt{;}$
    $\mathit{RetrieveVars}\ \mathit{ss}\ \mathit{nClass}\ \mathit{index}$
```

Rule A.46 Parametrised action call

$[\![N(e_1, \ldots, e_n)]\!]^{Action} = \texttt{N((}\mathit{JExp}\ e_1\texttt{),}\ldots\texttt{,(}\mathit{JExp}\ e_n\texttt{));}$

Rule A.47 Implicit parametrised action call

```
$[\![(\mathit{Decl} \bullet \mathit{Action})\,(e_1, \ldots, e_n)]\!]^{Action} =$
    $\mathit{DeclareActionClass}\ \mathit{Decl}\ \mathit{Action}\ \mathit{index}$
    $\texttt{I\_}\mathit{index}\ \texttt{i\_}\mathit{index\_index}\ \texttt{=}$
        $\texttt{new I\_}\mathit{index}\texttt{((}\mathit{JExp}\ e_1\texttt{),}\ldots\texttt{,(}\mathit{JExp}\ e_n\texttt{),}$
                    $\texttt{(}\mathit{JList}\ (\mathit{ListFirst}\ \mathit{LocalVarEnv})\texttt{));}$
    $\texttt{i\_}\mathit{index\_index}\texttt{.run();}$
    $\mathit{MergeLocalVars}\ \mathit{index}\ \mathit{index}\ L$
```

```
$DeclareActionClass$ : Decl ⇸ Action ⇸ ℕ ⇸ JCode
$DeclareActionClass\ Decl\ Action\ index$
    let $sep = $ if $LocalVarEnv = [\,]$ then $\epsilon$ else , in
        class I_$index$ implements CSProcess {
            $ParamsDecl\ Decl$
            $DeclLocalVars\ LocalVarEnv\ index\ L$
            public I_$index$(($ParamsArgs\ Decl$) $sep$
                            ($LocalVarsArg\ LocalVarEnv$)){
                $MultiAssign$ ($ParamsDecl\ Decl$) ($ParamsArgs\ Decl$)
                $InitLocalVars\ index\ L$
            }
            public void run (){
                $RenameVars$ ($[\![Action]\!]^{Action}$)
                            ($SetFirst\ LocalVarEnv$) $index\ L$
            }
        }
```

Rule A.48 Implicit recursive parametrised call

```
$[\![(\mu\,X \bullet (x_0 : T_0;\ \ldots;\ x_n : T_n \bullet Action(X(ec_0, \ldots, ec_n))))(ei_0, \ldots, ei_n)]\!]^{Action} =$
    let $ExtLocalVarEnv = LocalVarEnv \frown \langle (x_0, T_0), \ldots, (x_n, T_n)\rangle$ in
        class I_$index$ implements CSProcess {
            $DeclLocalVars\ ExtLocalVarEnv\ index\ L$
            public I_$index$($LocalVarsArg\ ExtLocalVarEnv$) {
                $InitLocalVars\ ExtLocalVarEnv\ index\ L$
             }
            public void run() {
                $RenameVars$
                    $[\![Action(($RunRecursion $index\ \langle ec_0, \ldots, ec_n\rangle$)$)]\!]^{Action}$
                    ($SetFirst\ ExtLocalVarEnv$) $index\ L$
            }
        };
        $RunRecursion\ index\ \langle ei_0, \ldots, ei_n\rangle$
```

Rule A.49 Iterated sequential composition

```
$[\![\fatsemi\ x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Action]\!]^{Action} =$
    $DeclareActionClass\ (x_1 : T_1;\ \ldots;\ x_n : T_n)\ Action\ index$
    for (($JType\ T_1$) x_1 = ($Min\ T_1$);
          x_1.compareTo(($Max\ T_1$))<=0; ($Inc\ x_1 : T_1$)){
        ...
        for (($JType\ T_n$) x_n = ($Min\ T_n$);
              x_n.compareTo(($Max\ T_n$))<=0; ($Inc\ x_n : T_n$)){
            I_$index$ i_$index$_$index$ =
                new I_$index$(($JExp\ e_1$),...,($JExp\ e_n$),
                            ($JList\ (ListFirst\ LocalVarEnv)$));
            i_$index$_$index$.run();
            $MergeLocalVars\ index\ index\ L$
        }
        ...
    }
```

Rule A.50 Iterated internal choice

```
$[\![\sqcap\ x_1 : T_1;\ \ldots;\ x_n : T_n \bullet Action]\!]^{Action} =$
    $ChooseIndexVars\ (x_1 : T_1;\ \ldots;\ x_n : T_n)$
    $DeclareActionClass\ (x_1 : T_1;\ \ldots;\ x_n : T_n)\ Action\ index$
    (new I_$index$(x_1,...,x_n)).run();
```

Commands

Rule A.51 Single assignment

$[\![x := e]\!]^{Action} =$ `x = (`$JExp\ e$`);`

Rule A.52 Multiple assignment

```
$[\![x_1, \ldots, x_n := e_1, \ldots, e_n]\!]^{Action} =$
    $\boldsymbol{if}\ (\{x_1, \ldots, x_n\} \cap (FV(e_1) \cup \ldots \cup FV(e_n)) = \varnothing)\ \boldsymbol{then}$
        x_1=($JExp\ e_1$); ...; x_n=($JExp\ e_n$);
    $\boldsymbol{else}$
        ($JType\ (CType\ x_1)$) aux_x_1 = ($JExp\ e_1$);
        ...;
        ($JType\ (CType\ x_n)$) aux_x_n = ($JExp\ e_n$);
        x_1=aux_x_1;
        ...;
        x_n=aux_x_n;
```

Rule A.53 If-command

$[\![\textbf{if } g_1 \rightarrow A_1 \,\Box\, \ldots \,\Box\, g_n \rightarrow A_n \textbf{ fi}]\!]^{Action} =$

```
    if((JExp g₁)){
        [[A₁]]^Action
    } else if (...) {
        ...
    } else if((JExp gₙ)){
        [[Aₙ]]^Action
    } else { while(true){} }
```

Rule A.54 Variable declaration

$[\![\textbf{var } x_1 : T_1; \ldots ; x_n : T_n \bullet Action]\!]^{Action} =$

```
    {
        (JType T₁) x_1;...; (JType Tₙ) x_n;
        [[Action]]^Action
    }
```

Rule A.55 Assumption

$[\![\{g\}]\!]^{Action} =$ `if((JExp g)){ (new Skip()).run(); } else { while(true){} }`

Z paragraphs in global scope

Rule A.56 Free type definition

$[\![\_]\!]^{FreeTypes} : \mathsf{Program} \rightarrow \mathsf{N} \rightarrow \mathsf{JCode}$

$[\![\epsilon]\!]^{FreeTypes}\ proj = \epsilon$

$[\![FTName ::= V_0 \mid \ldots \mid V_n\ FreeTypes]\!]^{FreeTypes}\ proj =$

```
    package proj.typing;
    public class FTName extends Type {
        DeclFTConstants (V₀ ⁀ ... ⁀ Vₙ) 0
        protected FTName(){}
        public FTName(int value) { this.setValue(value); }
    }
    [[FreeTypes]]^FreeTypes proj
```

$DeclFTConstants : \operatorname{seq} \mathsf{N} \rightarrow \mathbb{N} \rightarrow \mathsf{JCode}$

$DeclFTConstants\ \langle V \rangle \frown \langle\rangle\ n =$ `public static final int V = n;`
$\qquad DeclFTRange\ 0\ n$

$DeclFTConstants\ \langle V \rangle \frown VS)\ n =$ `public static final int V = n;`
$\qquad DeclFTConstants\ VS\ (n+1)$

$DeclFTRange : \mathbb{N} \rightarrowtail\!\!\!\!\!+ \mathbb{N} \rightarrowtail\!\!\!\!\!+ \mathsf{JCode}$
$DeclFTRange\ min\ max =$
    `public static final int MIN_VALUE =` min `;`
    `public static final int MAX_VALUE =` max `;`

Rule A.57 Global axiomatic definition

$[\![\_]\!]^{AxDefs} : \mathsf{Program} \rightarrowtail\!\!\!\!\!+ \mathsf{JCode}$
$[\![\epsilon]\!]^{AxDefs} = \epsilon$
$[\![v : T \mid v = e_1\ AxDefs]\!]^{AxDefs} =$
    `public static` $(JType\ T)\ v$`() { return` $(JExp\ e_1)$`; }`
    $[\![AxDefs]\!]^{AxDefs}$

Circus program

Rule A.58 *Circus* **program**

$[\![\_]\!]^{Program} : \mathsf{Program} \rightarrowtail\!\!\!\!\!+ \mathsf{N} \rightarrowtail\!\!\!\!\!+ \mathsf{JCode}$
$[\![FreeTypes\ AxDefs\ ChanDecls\ ProcDecls\,]\!]^{Program}\ proj =$
    $DeclareTypeClass\ (FreeTypes\ AxDefs\ ChanDecls\ ProcDecls)\ proj$
    $[\![FreeTypes\,]\!]^{FreeTypes}\ proj$
    $DeclareAxDefClass\ proj\ AxDefs$
    $[\![ProcDecls\,]\!]^{ProcDecls}\ proj$

```
$DeclareTypeClass : \mathsf{Program} \pfun \mathsf{N} \pfun \mathsf{JCode}$
$DeclareTypeClass\ prog\ proj =$
    package $proj$.typing;
    public abstract class Type {
        private int value;
        $DeclTypeConstants\ TypesEnv\ 0$
        public int getValue() { return this.value; }
        protected void setValue(int value) {
        this.value = value;
        }
        public boolean equals(Type other) {
            boolean equals = false;
            if (other != null) {
                boolean sameClass =
                    this.getClass().equals(other.getClass());
                boolean sameValue =
                    (this.getValue()==other.getValue());
                equals = (sameClass && sameValue);
            }
            return equals;
        }
        public int compareTo(Type other) {
            int compare = -1;
            if (other != null) {
                if (this.getValue() == other.getValue()) {
                    compare = 0;
                } else if (this.getValue() > other.getValue()) {
                    compare = 1;
                }
            }
            return compare;
        }
    }
```

```
$DeclareTypeConstants : \mathrm{seq}\,\mathsf{N} \pfun \mathbb{N} \pfun \mathsf{JCode}$
$DeclareTypeConstants\ \langle\rangle\ n =$
    public static final int MIN_TYPE_ID = 0;
    public static final int MAX_TYPE_ID = $n$;
$DeclareTypeConstants\ \langle T \rangle \frown TS\ n =$
    public static final int $(Capitals\ (JType\ T))$ = $n$;
    $DeclareTypeConstants\ TS\ (n + 1)$
```

```
$DeclareAxDefClass : \mathsf{Program} \pfun \mathsf{N} \pfun \mathsf{JCode}$
$DeclareAxDefClass\ AxDefs\ proj =$
    package $proj$.axiomaticDefinitions;
    import $proj$.typing.*;
    public class AxiomaticDefinitions { $[\![\, AxDefs \,]\!]^{AxDefs}$ }
```

Running the program

Rule A.59 Run process

$[\![]\!]^{Run} : \mathsf{Proc} \nrightarrow \mathsf{N} \nrightarrow \mathsf{JCode}$

$[\![N]\!]^{Run}\ proj =$

```
package proj;
import java.util.*;
import jcsp.lang.*;
import proj.axiomaticDefinitions.*;
import proj.processes.*;
import proj.typing.*;
import proj.util.*;
public class Main {
    public static void main(String args[]) { [[ N ]]^Proc }
}
```

Function *JExp*

JExp $a =$ `new FT(FT.a)`
  if a is an element of free type FT.
JExp $a =$ `new AxiomaticDefinitions.a()`
  if a is a global axiomatic definition.
JExp $a =$ `this.a()`
  if a is a local axiomatic defintion.
JExp $a =$ `a`
  if a is a process parameter, action parameter, state component or local variable.
JExp $a =$ `new CircusNumber(`a`)`
  if a is a number.
JExp $false =$ `false`
JExp $true =$ `true`
JExp $(\neg\ p) =$ `(!(`*JExp* p`))`
JExp $(p_1 \land\ p_2) =$ `(`*JExp* p_1 `&&` *JExp* p_2`)`
JExp $(p_1 \lor\ p_2) =$ `(`*JExp* p_1 `||` *JExp* p_2`)`
JExp $(p_1 \Leftrightarrow\ p_2) =$ `(`*JExp* p_1 `==` *JExp* p_2`)`
JExp $(p_1 \Rightarrow\ p_2) =$ `(!(`*JExp* p_1`)` `||` *JExp* p_2`)`

Appendix B

Refinement laws

B.1 Laws

Action systems

Law B.1 *Action system conversion*

$$\begin{array}{l}
\mu\, X \bullet \\
\quad a \rightarrow\ b \rightarrow\ X \\
\quad \Box \\
\quad c \rightarrow\ X \\
\\
\sqsubseteq \\
\\
\mathbf{var}\ pc := 0 \bullet \\
\quad \mu\, X \bullet \\
\qquad pc = 0\ \&\ a \rightarrow\ pc := 1;\ X \\
\qquad \Box \\
\qquad pc = 0\ \&\ c \rightarrow\ pc := 0;\ X \\
\qquad \Box \\
\qquad pc = 1\ \&\ b \rightarrow\ pc := 0;\ X
\end{array}$$

Law B.2 *Action system parallel*

$$\begin{array}{l}
[\![x]\!]\ i : I \bullet \\
\quad \left(\begin{array}{ll} \mu\, X \bullet & g_i\ \&\ x \rightarrow A_i;\ X \\ & \Box \\ & h_i\ \&\ y_i \rightarrow B_i;\ X \end{array} \right) \\
= \\
\mu\, X \bullet \\
\quad (\forall\, i : I \bullet g_i)\ \&\ x \rightarrow (|||\ i : I \bullet A_i);\ X \\
\quad \Box \\
\quad \Box\, i : I \bullet h_i\ \&\ y_i \rightarrow B_i;\ X
\end{array}$$

Law B.3 *Merge action system parallel*

$$\left(\begin{array}{l} \mu X \bullet \\ \qquad \Box\, i \bullet g_{1i} \mathrel{\&} comm_i \rightarrow A_{1i};\ X \\ \qquad \Box \\ \qquad \Box\, i \bullet h_{1i} \mathrel{\&} event_{1i} \rightarrow B_{1i};\ X \end{array}\right) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket \left(\begin{array}{l} \mu X \bullet \\ \qquad \Box\, i \bullet g_{2i} \mathrel{\&} comm_i \rightarrow A_{2i};\ X \\ \qquad \Box \\ \qquad \Box\, i \bullet h_{2i} \mathrel{\&} event_{2i} \rightarrow B_{2i};\ X \end{array}\right)$$

$$=$$

$$\begin{array}{l} \mu X \bullet \\ \qquad \Box\, i \bullet g_{1i} \wedge g_{2i} \mathrel{\&} comm_i \rightarrow (A_{1i} \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{2i});\ X \\ \qquad \Box \\ \qquad \Box\, i \bullet h_{1i} \mathrel{\&} event_{1i} \rightarrow B_{1i};\ X \\ \qquad \Box \\ \qquad \Box\, i \bullet h_{2i} \mathrel{\&} event_{2i} \rightarrow B_{2i};\ X \end{array}$$

provided

- $\forall\, i \bullet comm_i \in cs$
- $\forall\, i \bullet event_{1i}, event_{2i} \notin cs$ □

Law B.4 *Action system conversion 2*

$$\begin{array}{l} \mu X \bullet x \rightarrow y \rightarrow X \\ = \\ \mathbf{var}\ pc : \mathbb{N} \bullet \\ \qquad pc := 0; \\ \qquad \mu X \bullet pc = 0 \mathrel{\&} x \rightarrow pc := 1;\ X \\ \qquad\qquad \Box \\ \qquad\qquad pc = 1 \mathrel{\&} y \rightarrow pc := 0;\ X \end{array}$$

Law B.5 *Action system*

$$\begin{array}{l} \mu X \bullet g \mathrel{\&} x \rightarrow A;\ X \\ = \\ \mathbf{var}\ pc := 0 \bullet \\ \qquad \mu X \bullet (pc = 0 \wedge g) \mathrel{\&} x \rightarrow pc := 1;\ X \\ \qquad\qquad \Box \\ \qquad\qquad pc = 1 \mathrel{\&} A;\ pc := 0;\ X \end{array}$$

Law B.6 *Action system 2*

$$\begin{array}{l} \mu X \bullet \Box\, i \bullet g_i \mathrel{\&} x_i \rightarrow A_i(X) \\ = \\ \mathbf{var}\ pc := 0 \bullet \\ \qquad \mu X \bullet \Box\, i \bullet (pc = 0 \wedge g_i) \mathrel{\&} x_i \rightarrow pc := i;\ X \\ \qquad\qquad \Box \\ \qquad\qquad \Box\, i \bullet pc = i \mathrel{\&} A_i(pc := 0;\ X) \end{array}$$

where $i : 1 \mathinner{..} n$

Law B.7 *Action system 3*

$$
\begin{array}{l}
\textbf{var}\ pc := 0 \bullet \\
\quad \mu\, X \bullet \Box\, i \bullet (pc = e_i \wedge g_i)\ \&\ x_i \rightarrow A_i;\ X \\
\qquad \Box \\
\qquad (pc = e_{n+1}\ \&\ x_{n+1} \rightarrow B;\ A_{n+1};\ X \\
= \\
\textbf{var}\ pc := 0 \bullet \\
\quad \mu\, X \bullet \Box\, i \bullet (pc = e_i \wedge g_i)\ \&\ x_i \rightarrow A_i;\ X \\
\qquad \Box \\
\qquad (pc = e_{n+1}\ \&\ x_{n+1} \rightarrow pc := e_{n+2};\ X \\
\qquad \Box \\
\qquad (pc = e_{n+2}\ \&\ B;\ A_{n+1};\ X
\end{array}
$$

where *A_i are assignments, $i : 1 \mathinner{..} n$, $e_{n+2} \notin \{ i : 0 \mathinner{..} (n+1) \bullet e_i \}$*

Law B.8 *Eliminate useless branch*

$$
\begin{array}{l}
\quad \mu\, X \bullet g_1\ \&\ A_1;\ X \\
\qquad \Box \\
\qquad g_2\ \&\ A_2;\ X \\
\qquad \Box \\
\qquad g_3\ \&\ A_3;\ X \\
= \\
\quad \mu\, X \bullet g_1\ \&\ A_1;\ X \\
\qquad \Box \\
\qquad g_2\ \&\ A_2;\ X
\end{array}
$$

provided *$inv \Leftrightarrow \neg\, g_3$* □

Law B.9 *Merge branches 1*

$$
\begin{array}{l}
\{\neg\, g_2\};\ g_1\ \&\ x : [g_2];\ X \\
\Box \\
g_2\ \&\ A_2;\ x : [g_3];\ X \\
\Box \\
g_3\ \&\ x : [g_1];\ X \\
\Box \\
g_4\ \&\ A_4;\ X \\
\Box \\
g_5\ \&\ A_5;\ X \\
\Box \\
g_6\ \&\ \mathit{Skip} \\
\\
=
\end{array}
$$

$\{\neg\, g_2\};\ g_1\ \&\ x : [g_2];\ X$
$\Box$
$g_1\ \&\ A_2;\ x : [g_1];\ X$
$\Box$
$g_2\ \&\ A_2;\ x : [g_3];\ X$
$\Box$
$g_4\ \&\ A_4;\ X$
$\Box$
$g_5\ \&\ A_5;\ X$
$\Box$
$g_6\ \&\ \mathit{Skip}$

provided

- $g_1 \Rightarrow \neg\, g_6$
- g_1, g_2, g_3 *and* g_4 *are mutually exclusive*
- $FV(A_5) \cup FV(g_5) \neq FV(g_1) \cup FV(g_2) \cup FV(g_3) \cup x \cup FV(A_2) \cup FV(A_4)$ $\Box$

Law B.10 *Merge branches 2*

$\mu\, X \bullet$
$\quad \Box\, i \bullet g_{1i}\ \&\ A_1;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X$
$\quad \Box$
$\quad \Box\, i \bullet g_{2i}\ \&\ A_2;\ x : [\neg\, g_{2i} \wedge \neg\, g_{2i}];\ X$
$\quad \Box$
$\quad \Box\, i \bullet g_{1i}\ \&\ c \rightarrow A_3;\ x : [g_{2i} \wedge \neg\, g_{1i}];\ X$
$\quad \Box$
$\quad (\neg\, \exists\, i \bullet g_{1i} \vee g_{2i})\ \&\ B$
$=$
$\mu\, X \bullet$
$\quad \Box\, i \bullet g_{1i}\ \&\ c \rightarrow A_3;\ x : [g_{2i} \wedge \neg\, g_{1i}];\ X$
$\quad \Box$
$\quad \mathit{Skip}$
$\mu\, X \bullet$
$\quad \Box\, i \bullet g_{1i}\ \&\ A_1;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X$
$\quad \Box$
$\quad \Box\, i \bullet g_{2i}\ \&\ A_2;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X$
$\quad \Box$
$\quad (\neg\, \exists\, i \bullet g_{1i} \vee g_{2i})\ \&\ B$

Law B.11 *Merge branches 3*

$\{\neg\, \exists\, i \bullet g_{3i}\}\ \mu\, X \bullet$
$\quad \Box\, i \bullet g_{1i}\ \&\ A_1;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i} \wedge g_{3i}];\ X$
$\quad \Box$
$\quad \Box\, i \bullet g_{2i}\ \&\ A_2;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X$
$\quad \Box$
$\quad \Box\, i \bullet g_{3i}\ \&\ A_3;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X$
$\quad \Box$
$\quad (\neg\, \exists\, i \bullet g_{1i} \vee g_{2i})\ \&\ B$

$$
\begin{array}{l}
= \\
\mu X \bullet \\
\quad \Box\, i \bullet g_{1i} \mathbin{\&} A_1;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i} \wedge g_{3i}];\ X \\
\quad \Box \\
\quad \Box\, i \bullet g_{2i} \mathbin{\&} A_2;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X \\
\quad \Box \\
\quad (\neg\, \exists\, i \bullet g_{1i} \vee g_{2i}) \mathbin{\&} B \\
\mu X \bullet \\
\quad \Box\, i \bullet g_{3i} \mathbin{\&} A_3;\ x : [\neg\, g_{1i} \wedge \neg\, g_{2i}];\ X \\
\quad \Box \\
\quad \mathit{Skip}
\end{array}
$$

Internal actions

Law B.12 *External choice/Internal choice - internal action*

$$((g_1 \mathbin{\&} A) \Box (g_2 \mathbin{\&} B)) \sqcap (g_2 \mathbin{\&} B) = ((g_1 \mathbin{\&} A) \Box (g_2 \mathbin{\&} B))$$

provided

- *B contains only internal actions;*
- $g_2 = \mathit{true}$ □

Law B.13 *External choice X internal choice for internal action*

$$
\begin{array}{l}
\Box\, i \bullet g_i \mathbin{\&} c_i \rightarrow A_i \\
\Box \\
\Box\, i \bullet h_i \mathbin{\&} x_i := e_i \\
= \\
\left(\begin{array}{l}
\Box\, i \bullet g_i \mathbin{\&} c_i \rightarrow A_i \\
\Box \\
\Box\, i \bullet h_i \mathbin{\&} x_i := e_i
\end{array} \right) \\
\sqcap \\
\Box\, i \bullet h_i \mathbin{\&} x_i := e_i
\end{array}
$$

provided $\vee\, i \bullet h_i$. □

Law B.14 *Merge internal action*

$$
\begin{array}{l}
\{\neg\, g_3\} \\
\mu X \bullet \\
\quad g_1 \mathbin{\&} A_1;\ [\neg\, g_3];\ X \\
\quad \Box \\
\quad g_2 \mathbin{\&} A_2;\ [\neg\, g_3];\ X \\
=
\end{array}
$$

$$\begin{array}{l}
\{\neg\, g_3\} \\
\mu\, X \bullet \\
\quad g_1 \;\&\; x : [true, g_3 \wedge \neg\, g_1];\; X \quad (1) \\
\quad \Box \\
\quad g_3 \;\&\; A_1;\; [\neg\, g_3];\; X \quad (2) \\
\quad \Box \\
\quad g_2 \;\&\; A_2;\; [\neg\, g_3];\; X \quad (3)
\end{array}$$

provided

- $wrtV(A_2) \cap FV(g_3) = \varnothing$
- $wrtV(A_2) \cap FV(g_1) = \varnothing$
- $x \cap FV(g_2) = \varnothing$
- $\exists\, x' \bullet g_3 \neg \;\vee g_1$ □

Law B.15 *Elimination of internal action 2*

$$\begin{array}{l}
\{pc = x\} \\
\quad \mu\, X \bullet \\
\quad\quad pc = x \;\&\; pc := e;\; X \\
\quad\quad \Box \\
\quad\quad f(pc = e) \;\&\; A;\; X \\
\quad\quad \Box \\
\quad\quad pc = y \;\&\; B;\; X \\
= \\
\{pc = x\} \\
\quad \mu\, X \bullet \\
\quad\quad f(pc = x) \;\&\; pc := e;\; A;\; X \\
\quad\quad \Box \\
\quad\quad pc = y \;\&\; B;\; X
\end{array}$$

provided $pc = e \Rightarrow g_1 \wedge \neg\, g_2$ □

Law B.16 *Interleaving/Sequential composition - internal actions*

$$(A \;[\![ns_1 \mid ns_2]\!]\; B) = A;\; B$$

provided

- *A and B contain only internal actions*
- $usedV(A) \cap ns_2 = \varnothing$
- $usedV(B) \cap ns_1 = \varnothing$ □

Law B.17 *Interleaving/Sequential composition - internal actions 2*

$$(\,|||\; i : T \;[\![x_i]\!] \bullet x_i := e_i) = (\,⨟\; i : T \bullet x_i := e_i)$$

Hiding

Law B.18 *Hiding conditional external choice distribution 2*

$$\left(\begin{array}{l} b \,\&\, x \to P \\ \Box \\ c \,\&\, y \to Q \end{array}\right) \setminus \{x\} = \left(\begin{array}{l} b \,\&\, (P \setminus \{x\} \Box (\mathit{Stop} \sqcap c \,\&\, y \to Q \setminus \{x\})) \\ \Box \\ \neg\, b \wedge c \,\&\, y \to Q \setminus \{x\} \end{array}\right)$$

Law B.19 *Hiding conditional external choice distribution 2a*

$$\left(\begin{array}{l} b \,\&\, x \to P \\ \Box \\ c \,\&\, Q \end{array}\right) \setminus \{x\} = \left(\begin{array}{l} b \,\&\, (P \setminus \{x\} \Box (\mathit{Stop} \sqcap c \,\&\, Q \setminus \{x\})) \\ \Box \\ \neg\, b \wedge c \,\&\, Q \setminus \{x\} \end{array}\right)$$

Law B.20 *Hiding conditional external choice distribution 3*

$$\left(\begin{array}{l} g_a \,\&\, a \to A \\ \Box \\ g_b \,\&\, b \to B \\ \Box \\ g_c \,\&\, c \to C \\ \Box \\ g_d \,\&\, d \to D \end{array}\right) \setminus \{a, b\}$$

$$=$$

$$g_a \vee g_b \,\& \left(\begin{array}{l} \left(\begin{array}{l} g_c \,\&\, c \to C \\ \Box \\ g_d \,\&\, d \to D \end{array}\right) \sqcap \mathit{Stop} \\ \Box \\ \left(\begin{array}{l} (g_a \wedge g_b) \,\&\, A \setminus \{a, b\} \sqcap B \setminus \{a, b\} \\ \Box \\ (g_a \wedge \neg\, g_b) \,\&\, A \setminus \{a, b\} \\ \Box \\ (\neg\, g_a \wedge g_b) \,\&\, B \setminus \{a, b\} \end{array}\right) \quad (*) \end{array}\right)$$

$$\Box$$

$$\neg\,(g_a \vee g_b) \wedge (g_c \vee g_d) \,\& \left(\begin{array}{ll} g_c \,\&\, c \to C & \text{(A)} \\ \Box & \\ g_d \,\&\, d \to D & \text{(B)} \end{array}\right)$$

Obs: In case $A \setminus \{a, b\}$ and $B \setminus \{a, b\}$ contain only internal actions, then

$$\begin{array}{l} (g_a \wedge g_b) \,\&\, A \setminus \{a, b\} \sqcap B \setminus \{a, b\} \\ \Box \\ (g_a \wedge \neg\, g_b) \,\&\, A \setminus \{a, b\} \\ \Box \\ (\neg\, g_a \wedge g_b) \,\&\, B \setminus \{a, b\} \\ = \\ g_a \,\&\, A \setminus \{a, b\} \\ \Box \\ g_b \,\&\, B \setminus \{a, b\} \end{array}$$

This happens because $A \Box A = A \sqcap A$ if A contains only internal events.

Law B.21 *Hiding identity*

$$A \setminus cs = A$$

provided $cs \cap usedC(A) = \varnothing$ $\Box$

Law B.22 *Hiding/variable declaration - distribution*

$$(\mathbf{var}\ x \bullet A) \setminus cs = \mathbf{var}\ x \bullet A \setminus cs$$

Law B.23 *Hiding/sequence - distribution*

$$(A;\ B) \setminus cs = (A \setminus cs;\ B \setminus cs)$$

Law B.24 *Hiding/recursion - distribution*

$$(\mu X \bullet F(X)) \setminus cs = (\mu X \bullet F(X) \setminus cs)$$

Law B.25 *Hiding/interleaving - distribution*

$$(|||\ i : I \bullet A_i) \setminus cs = |||\ i : I \bullet (A_i \setminus cs)$$

Law B.26 *Hiding/assignment - distribution*

$$(x := e) \setminus cs = (x := e)$$

Recursion

Law B.27 *Recursion halt*

$$\begin{array}{l} \mu X \bullet A;\ X \\ \qquad\quad \Box \\ \qquad\quad g \mathbin{\&} x := e;\ B \end{array}$$

$$=$$

$$\left(\begin{array}{l} \mu X \bullet A;\ X \\ \qquad\quad \Box \\ \qquad\quad g \mathbin{\&} Skip \end{array} \right);\ x := e;\ B$$

Law B.28 *Elimination of redundant branch in recursion*

```
$\mu X \bullet$
    $g_1 \mathbin{\&} A_1;\ X$
    $\Box$
    $g_2 \mathbin{\&} A_2;\ X$
    $\Box$
    $p \mathbin{\&}$
        $\mu Y \bullet$
            $g_1 \mathbin{\&} A_1;\ Y$
            $\Box$
            $B$
$=$
$\mu X \bullet$
    $g_1 \mathbin{\&} A_1;\ X$
    $\Box$
    $g_2 \mathbin{\&} A_2;\ X$
    $\Box$
    $p \mathbin{\&} B$
```

provided $\neg\,(p \Rightarrow \neg\, g_1)$ $\Box$

Law B.29 *Assumption/recursion - refinement*

$$\{p\};\ \mu\ X \bullet F(X) \sqsubseteq G(\{p\};\ \mu\ X \bullet F(X)) \Rightarrow \{p\};\ \mu\ X \bullet F(X) \sqsubseteq \mu X \bullet G(X)$$

Law B.30 *Useless assignment - recursion*

$$x := e;\ \mu X \bullet F(X) = \mu X \bullet x := e;\ F(X)$$

provided *provided that* $x \notin FV(F(X))$. $\Box$

Law B.31 *Fixed point rolling*

$$G(\mu X \bullet F(G(X))) = \mu X \bullet G(F(X))$$

Law B.32 *Fixed point diagonal*

$$\mu X \bullet F(X, X) = \mu X \bullet (\mu X \bullet F(X, Y))$$

Law B.33 *Least fixed point*

$$F(Y) \sqsubseteq Y \Rightarrow \mu X \bullet F(X) \sqsubseteq Y$$

Law B.34 *Recursion split*

```
{$b$}
$\mu X \bullet$
    $(b \wedge g_1) \mathbin{\&} A_1;\ X$
    $\Box$
    $(b \wedge g_2) \mathbin{\&} A_2;\ x : [\neg\ b];\ X$
    $\Box$
    $(\neg\ b \wedge g_3) \mathbin{\&} A_3;\ X$
    $\Box$
    $(\neg\ b \wedge g_4) \mathbin{\&} A_4;\ x : [b];\ X$
    $\Box$
    $g_5 \mathbin{\&} A_5;\ X$

$=$

{$b$}
$\mu X \bullet$
    $(b \wedge g_1) \mathbin{\&} A_1;\ X$
    $\Box$
    $(b \wedge g_2) \mathbin{\&} A_2;\ x : [\neg\ b]$
    $\Box$
    $g_5 \mathbin{\&} A_5;\ X$
; $\mu X \bullet$
    $(\neg\ b \wedge g_3) \mathbin{\&} A_3;\ X$
    $\Box$
    $(\neg\ b \wedge g_4) \mathbin{\&} A_4;\ x : [b]$
    $\Box$
    $g_5 \mathbin{\&} A_5;\ X$
```

provided $FV(b) \cap wrtV(A_1, A_2, A_3, A_4, A_5) = \varnothing$ $\Box$

Law B.35 *Parametrised/Non-parametrised recursion*

$$(\mu X \bullet p : T \bullet F(X(e_1)))(e_2)$$
$$=$$
$$\mathbf{var}\ p : T \bullet p := e_2 \bullet \mu X \bullet F(p := e_1;\ X)$$

Law B.36 *Useless variable - recursion*

$$
\begin{array}{l}
\{pc = e\};\\
\mu X \bullet\\
\quad \Box\, i \bullet (pc = e \land g_i) \mathbin{\&} A_i;\ pc := e;\ X\\
\quad \Box\\
\quad (pc = e \land g) \mathbin{\&} \mathit{Skip}\\
\quad \Box\\
\quad \Box\, i \bullet h_i \mathbin{\&} B_i;\ X\\
=\\
\{pc = e\};\\
\mu X \bullet\\
\quad \Box\, i \bullet g_i \mathbin{\&} A_i;\ X\\
\quad \Box\\
\quad g \mathbin{\&} \mathit{Skip}\\
\quad \Box\\
\quad \Box\, i \bullet h_i \mathbin{\&} B_i;\ X
\end{array}
$$

provided

- A_i *contains only internal actions*
- $pc \notin FV(A_i) \cup FV(B_i)$ □

Law B.37 *Useless Recursion*

$$\mu X \bullet A = A$$

provided A *does not contain a recursive call to* X □

Iteration

Law B.38 *Iteration*

$$
\begin{array}{l}
\mu X \bullet\\
\quad \Box\, i \bullet h_i \mathbin{\&} A_i;\ X\\
\quad \Box\\
\quad \Box\, i \bullet (g_i \land \mathit{variant} > 0) \mathbin{\&} x : [\mathit{inv} \land g_i, \mathit{inv} \land \mathit{variant}' < \mathit{variant}];\ X\\
\quad \Box\\
\quad \mathit{variant} = 0 \mathbin{\&} A\\
\\
\sqsubseteq\\
\\
\mu X \bullet\\
\quad \Box\, i \bullet h_i \mathbin{\&} A_i;\ X\\
\quad \Box\\
\quad x : [\mathit{inv}, \mathit{inv} \land (\land\, i \bullet \neg\, g_i)];\ A
\end{array}
$$

Alternation

Law B.39 *Alternation introduction*

$$w : [pre, pos]$$
$$=$$
$$\mathbf{if}(\| i \bullet G_i \rightarrow w : [G_i \wedge pre, pos])\mathbf{fi}$$

provided $pre \Rightarrow (\vee\ i \bullet G_i)$ $\Box$

Law B.40 *Alternation Introduction*

$$w : [pre, post] \sqsubseteq \mathbf{if}\ \|_i g_i \rightarrow\ w : [g_i \wedge pre, post]\ \mathbf{fi}$$

provided $pre \Rightarrow \bigvee_i g_i$ $\Box$

Law B.41 *Alternation/Guarded Actions - interchange*

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

provided

- $g_1 \vee g_2$
- $g_1 \Rightarrow \neg\ g_2$ $\Box$

Sequential composition

Law B.42 *Sequential composition*

$$w : [pre, pos] = w : [pre, mid];\ w : [mid, pos]$$

provided *mid and pos do not contain initial variables* $\Box$

Law B.43 *Sequence unit*

$$A = A;\ Skip = Skip;\ A$$

Assignment

Law B.44 *Assignment introduction*

$$x : [x = e] = x := e$$

Law B.45 *Assignment introduction*

$$\{pre\};\ w : [pre, pos] = \{pre\};\ w := e$$

provided $pre \Rightarrow pos[w/e]$ □

Law B.46 *Assignment introduction - assumption*

$$\{x = e\} = \{x = e\};\ x := e$$

Law B.47 *Switch assignment*

$$\{x := e_1\};\ w, x : [pre, post \wedge x = e_1 \Rightarrow x = e_2];\ A(x = e_2)$$

$$=$$

$$\{x := e_1\};\ A(x = e_1)w, x : [pre, post \wedge x = e_1 \Rightarrow x = e_2];$$

Law B.48 *Move assignment*

$$x := e;\ A = A;\ x := e$$

provided $x \notin FV(A)$ □

Law B.49 *Useless assignment 3*

$$(x := e;\ x := f) = (x := f)$$

provided *x is not free in f.* □

Assumption

Law B.50 *Assumption introduction*

$$\{p\};\ A \sqsubseteq A$$

Law B.51 *Assumption introduction - postcondition*

$$w : [pre, pos] = w : [pre, pos];\ \{pos[w/w']\}$$

provided *pos does not contain any reference to initial variables.* □

Law B.52 *Assumption introduction - assumption*

$$\{g\} = \{g\};\ \{g_1\}$$

provided $g \Rightarrow g_1$ $\Box$

Law B.53 *Assumption introduction - assignment*

$$x := e = x := e;\ \{x = e\}$$

Law B.54 *Assumption move - assignment*

$$\{x_1 = e_1\}x_2 := e_2 = x_2 := e_2;\ \{x_1 = e_1\}$$

Law B.55 *Assumption/Recursion*

$$\begin{array}{l}\{g\};\ \mu X \bullet F(X)\\ =\\ \{g\};\ \mu X \bullet (\{g\};\ F(X))\end{array}$$

provided $\{g\};\ F(X) \sqsubseteq F(\{g\};\ X)$ $\Box$

Law B.56 *Assumption/Guard - introduction*

$$\{g\};\ A = \{g\};\ g \mathbin{\&} A$$

Law B.57 *Guard/Assumption - introduction 1*

$$g \mathbin{\&} A = g \mathbin{\&} \{g\};\ A$$

Law B.58 *Assumption/External choice - distribution*

$$\{g\};\ (A_1 \Box A_2) = (\{g\};\ A_1) \Box (\{g\};\ A_2)$$

Law B.59 *Assumption/Prefix - distribution*

$$\{g\};\ c \rightarrow A \sqsubseteq c \rightarrow \{g\};\ A$$

Law B.60 *Assumption/Prefix - distribution 2*

$$\{g\};\ c \rightarrow A = \{g\};\ c \rightarrow \{g\};\ A$$

Specification statement

Law B.61 *Equivalent post-condition*

$$w : [pre, pos] = w : [pre, pos_1]$$

provided $pre \wedge post_1[w/w'] \Leftrightarrow pos[w/w']$ □

Law B.62 *Unchanged variable*

$$w : [pre, f(y)] = w : [pre, f(y')]$$

provided $y \notin w$. □

Law B.63 *Move specification statement*

$$\begin{array}{l} \{x = e\}; \\ x : [x = e \Rightarrow x = f]; \\ A(x = f) \\ = \\ \{x = e\}; \\ A(x = e) \\ x : [x = e \Rightarrow x = f]; \end{array}$$

Prefixing

Law B.64 *Prefix/Skip*

$$c \rightarrow A = (c \rightarrow Skip) ; A$$

$$c.e \rightarrow A = (c.e \rightarrow Skip) ; A$$

Guards

Law B.65 *Guard combination*

$$g_1 \,\&\, (g_2 \,\&\, A) = (g_1 \wedge g_2) \,\&\, A$$

Law B.66 *Guard expansion*

$$g_1 \,\&\, A \,\Box\, g_2 \,\&\, A = (g_1 \vee g_2) \,\&\, A$$

Law B.67 *Guard/External choice - distribution*

$$g \,\&\, (A_1 \,\Box\, A_2) = (g \,\&\, A_1) \,\Box\, (g \,\&\, A_2)$$

Law B.68 *Guard/Internal choice - distribution*

$g \mathbin{\&} (A_1 \sqcap A_2) = (g \mathbin{\&} A_1) \sqcap (g \mathbin{\&} A_2)$

Law B.69 *True guard*

$\mathit{true} \mathbin{\&} A = A$

Law B.70 *False guard*

$\mathit{false} \mathbin{\&} A = \mathit{Stop}$

Law B.71 *Guarded Stop*

$g \mathbin{\&} \mathit{Stop} = \mathit{Stop}$

External choice

Law B.72 *External choice commutativity*

$A_1 \Box A_2 = A_2 \Box A_1$

Law B.73 *External choice/Sequence - distribution*

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

Law B.74 *External choice/Sequence - distribution 2*

$((g_1 \mathbin{\&} A_1) \Box (g_2 \mathbin{\&} A_2));\ B = ((g_1 \mathbin{\&} A_1);\ B) \Box ((g_2 \mathbin{\&} A_2);\ B)$

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

Law B.75 *External choice/Internal choice - distribution*

$A \Box (B \sqcap C) = (A \Box B) \sqcap (A \Box C)$

Law B.76 *External choice unit*

$\mathit{Stop} \Box A = A$

Law B.77 *External choice/Guarded action*

$(g_1 \mathbin{\&} A_1) \Box (g_2 \mathbin{\&} A_2) = (g_1 \lor g_2) \mathbin{\&} ((g_1 \mathbin{\&} A_1) \Box (g_2 \mathbin{\&} A_2))$

Law B.78 *Useless assignment - external choice*

$x := e;$
$g_1 \mathbin{\&} c_1 \rightarrow A_1$
$\Box$
$g_2 \mathbin{\&} A_2$

$=$

$g_1 \mathbin{\&} c_1 \rightarrow x := e;\ A_1$
$\Box$
$g_2 \mathbin{\&} A_2$

$=$

$g_1 \mathbin{\&} c_1 \rightarrow A_1$
$\Box$
$g_2 \mathbin{\&} x := e;\ A_2$

provided $x \notin FV(g_1)$ *and* $x \notin FV(g_2)$. $\Box$

Law B.79 *Internal choice/External choice/Stop*

$(A \sqcap Stop) \Box B$
$=$
$(A \Box B) \sqcap (Stop \Box B)$
$=$
$(A \Box B) \sqcap B$

Law B.80 *Common branch external choice*

$g_1 \mathbin{\&} (A \Box B)$
$\Box$
$g_2 \mathbin{\&} (A \Box C)$

$=$

A
$\Box$
$g_1 \mathbin{\&} B$
$\Box$
$g_2 \mathbin{\&} C$

provided $(g_1 \vee g_2) \Leftrightarrow true$ $\Box$

Internal choice

Law B.81 *Internal choice elimination*

$$A \sqcap A = A$$

Law B.82 *Internal choice commutativity*

$$A \sqcap B = B \sqcap A$$

Variable block

Law B.83 *Useless assignment*

$$(\mathbf{var}\ x : T \bullet A;\ x := e) = (\mathbf{var}\ x : T \bullet A)$$

Law B.84 *Useless assignment 2*

$$(\mathbf{var}\ x : T \bullet x := e;\ A) = (\mathbf{var}\ x : T \bullet A)$$

provided $x \notin FV(A)$ □

Law B.85 *Variable block/sequence - distribution*

$$A_1;\ (\mathbf{var}\ x : T \bullet A_2);\ A_3 = (\mathbf{var}\ x : T \bullet A_1;\ A_2;\ A_3)$$

provided $\{x, x'\} \cap (FV(A_1) \cup FV(A_3)) = \varnothing$ □

Law B.86 *Join variable blocks*

$$\mathbf{var}\ x : T_1;\ y : T_2 \bullet A = \mathbf{var}\ x : T_1 \bullet \mathbf{var}\ y : T_2 \bullet A$$

Law B.87 *Unused variable*

$$(\mathbf{var}\ x : T \bullet A) = A$$

provided $\{x, x'\} \cap FV(A) = \varnothing$ □

Parallelism

Law B.88 *Parallel state*

$$\parallel i : I \bullet (\mathbf{var}\ x : T \bullet P_i) = \mathbf{var}\ x_i : T \bullet (\ \parallel i : I \bullet P_i[x_i/x])$$

Law B.89 *Parallel assignment*

$$\parallel i : I \bullet (x_i := e;\ P_i) = (\interleave i : I \bullet x_i := e);\ (\ \parallel i : I \bullet P_i)$$

Law B.90 *Parallelism/Interleaving - equivalence*

$$A \ [\![ns_2 \mid ns_2]\!]\ B = A \ [\![ns_2 \mid \varnothing \mid ns_2]\!]\ B$$

Law B.91 *Parallelism/Interleaving - equivalence 2*

$$A \ [\![ns_2 \mid ns_2]\!]\ B = A \ [\![ns_2 \mid cs \mid ns_2]\!]\ B$$

provided

- $usedC(A) \cap cs = \varnothing$
- $usedC(B) \cap cs = \varnothing$ $\square$

Simulation

Law B.92 *Simple prefix distribution*

$$c.ae \rightarrow Skip \preceq c.ce \rightarrow Skip$$

provided $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow ae = ce$ $\square$

Law B.93 *Guard distribution*

$$ag \ \&\ A_1 \preceq cg \ \&\ A_2$$

provided

- $\forall P_1.st;\ P_2.st;\ L \bullet R \Rightarrow (ag \Leftrightarrow cg)$
- $A_1 \preceq A_2$ $\square$

Law B.94 *Sequence distribution*

$$A_1;\ A_2 \preceq B_1;\ B_2$$

provided

- $A_1 \preceq B_1$
- $A_2 \preceq B_2$ $\square$

Law B.95 *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 injective function from the abstract to the concrete state.* $\Box$

Law B.96 *Interleave distribution*

$$A_1 \mathbin{|\!|\![} ns_1 \mid ns_2 \mathbin{]\!|\!|} A_2 \preceq B_1 \mathbin{|\!|\![} ns_1 \mid ns_2 \mathbin{]\!|\!|} B_2$$

provided

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

Law B.97 *Recursion distribution*

$$\mu X \bullet F_A(X) \preceq \mu X \bullet F_C(X)$$

provided $F_A \preceq F_C$ $\Box$

B.2 Lemmas

Lemma B.98 *Converting a client into an action system*

```
$\mu X \bullet$
    $to_A!i \rightarrow$
        $from_A.i?any \rightarrow$
            $to_B!i \rightarrow$
                $from_B.i?synchronised \rightarrow$
                    $(synchronised = true) \;\&\; q.i \rightarrow\; X$
                    $\Box$
                    $(synchronised = false) \;\&\; X$
        $\Box$
        $interrupt.i \rightarrow$
            $to_A!(flip\ \ i) \rightarrow X$
            $\Box$
            $from_A.i?anyt \rightarrow to_B!(flip\ \ i) \rightarrow from_B.i?any \rightarrow X$
```

$=$

```
$\mathbf{var}\ pc := 0, sync \bullet$
    $\mu X \bullet$
        $pc = 0 \;\&\; to_A!i \rightarrow pc := 1;\; X$
        $\Box$
        $pc = 1 \;\&\; from_A.i?any \rightarrow pc := 2;\; X$
        $\Box$
        $pc = 2 \;\&\; to_B!i \rightarrow pc := 3;\; X$
        $\Box$
        $pc = 3 \;\&\; from_B.i?synchronised \rightarrow pc := 4;\; sync := synchronised;\; X$
        $\Box$
        $(pc = 4 \wedge sync = true) \;\&\; q.i \rightarrow pc := 0;\; X$
        $\Box$
        $(pc = 4 \wedge sync = false) \;\&\; pc := 0;\; X$
        $\Box$
        $pc = 1 \;\&\; interrupt.i \rightarrow pc := 5;\; X$
        $\Box$
        $pc = 5 \;\&\; to_A!(flip\ \ i) \rightarrow pc := 0;\; X$
        $\Box$
        $pc = 5 \;\&\; from_A.i?anyt \rightarrow pc := 7;\; X$
        $\Box$
        $pc = 7 \;\&\; to_B?(flip\ \ i) \rightarrow pc := 8;\; X$
        $\Box$
        $pc = 8 \;\&\; from_B.i?anyf \rightarrow pc := 0;\; X$
```

Lemma B.99 *Converting the controller into an action system*

```
**var** $count_x : I \bullet count_x := n;$
$\mu X \bullet$
    $(count_x > 0 \wedge count_x \leq n)$ &
        $to?nextOffer \rightarrow$
            $(nextOffer \geq 0)$ & $count_x := count_x - 1;\ X$
            $\Box$
            $(nextOffer < 0)$ & $count_x := count_x + 1;\ X$
    $\Box$
    $(count_x = 0)$ &
        **var**$i_y : I \bullet i_y := 0;$
        $\mu Y \bullet$
            $(i_y < n)$ & $from.i_y!true \rightarrow i_y := i_y + 1;\ Y$
            $\Box$
            $(i_y = n)$ &
                **var** $i_z, count_z : I \bullet i_z := 0;\ count_z := n;$
                $\mu Z \bullet$
                    $(i_z \geq 0 \wedge i_z < n)$ & $to?nextcommit \rightarrow$
                        $(nextcommit \geq 0)$ & $i_z := i_z + 1;\ count_z := count_z - 1;\ Z$
                        $\Box$
                        $(nextcommit < 0)$ & $i_z := i_z + 1;\ Z$
                    $\Box$
                    $(i = n)$ &
                        **var** $i_w : I \bullet i_w := 0;$
                        $\mu W \bullet$
                            $(i_w < n)$ & $from.i_w!(count_z = 0) \rightarrow\ i_w := i_w + 1;\ W$
                            $\Box$
                            $(i_w = n)$ & $count_x := n;\ X$
```

$=$

```
**var** $count_x, i_y, i_z, count_z, i_w, pc, nextO, nextC \bullet$
$count_x := n;\ pc := 0;$
    $\mu X \bullet$
        $(pc = 0 \wedge count_x > 0)$ & $to?nextOffer \rightarrow pc := 1;\ nextO := nextOffer;\ X$
        $\Box$ $(pc = 1 \wedge nextO \geq 0)$ & $pc := 0;\ count_x := count_x - 1;\ X$
        $\Box$ $(pc = 1 \wedge nextO < 0)$ & $pc := 0;\ count_x := count_x + 1;\ X$
        $\Box$ $(pc = 0 \wedge count_x = 0)$ & $pc := 2;\ i_y := 0;\ X$
        $\Box$ $(pc = 2 \wedge i_y < n)$ & $from.i!true \rightarrow pc := 2;\ i_y := i_y + 1;\ X$
        $\Box$ $(pc = 2 \wedge i_y = n)$ & $pc := 3;\ i_z := 0;\ count_z := n;\ X$
        $\Box$ $(pc = 3 \wedge i_z \geq 0 \wedge i_z < n)$ & $to?nextCommit \rightarrow pc := 4;\ nextC := nextCommit;\ X$
        $\Box$ $(pc = 4 \wedge nextC \geq 0)$ & $pc := 3;\ i_z := i_z + 1;\ count_z := count_z - 1;\ X$
        $\Box$ $(pc = 4 \wedge next < 0)$ & $pc := 3;\ i_z := i_z + 1;\ X$
        $\Box$ $(pc = 3 \wedge i_z = n)$ & $pc := 5;\ i_w = 0;\ X$
        $\Box$ $(pc = 5 \wedge i_w < n)$ & $from.i!(count_z = 0) \rightarrow pc := 5;\ i_w := i_w + 1;\ X$
        $\Box$ $(pc = 5 \wedge i_w = n)$ & $pc := 0;\ count_x := n;\ X$
```

Bibliography

[1] CZT - Community Z Tools Website. http://czt.sourceforge.net/.

[2] Java Development Kit. http://java.sun.com/javase/.

[3] JUnit.org. http://www.junit.org.

[4] Velocity Webpage. http://jakarta.apache.org/velocity/.

[5] J. R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[6] P. Austin. Java Communicating Sequential Processes – Design of JCSP Language Classes. BSc in Computer Science 3rd Year Project, University of Kent at Canterbury, May 1998. http://www.cs.kent.ac.uk/projects/ofa/jcsp0-5/design/langdes.pdf.

[7] A. Barboza and A. L. C. Cavalcanti. Ferramenta para *Circus*: Analisador Sintático – Relatório de Atividades do Bolsista CNPq, Centro de Informática, Universidade Federal de Pernambuco, Brazil, August 2002.

[8] G. Booch, I. Jacobsen, and J. Rumbaugh. *Unifying Modeling Language User Guide*. Addison-Wesley, 1997.

[9] M. J. Butler. csp2B: A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12(3):182–198, 2000.

[10] D. A. Carrington, D. J. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: an Object-Oriented Extension to Z. In *Formal Description Techniques (FORTE '89)*, pages 281–296. North-Holland Publishing Co., 1989.

[11] 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, *Formal Methods (FM 2005)*, volume 3582 of *LCNS*, pages 253–268. Springer-Verlag, 2005.

[12] A. L. C. Cavalcanti and A. C. A. Sampaio. From CSP-OZ to Java with Processes. In *16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, page 161. IEEE Computer Society, 2002.

[13] 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.

[14] C. Fischer. CSP-OZ: a Combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438. Chapman and Hall, London, 1997.

[15] Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.28*, 1999.

[16] A. Freitas. From *Circus* to Java: Implementation and Verification of a Translation Strategy – MSc Thesis Additional Material, December 2005. http://www.cs.york.ac.uk/circus/jcsp/freitas-msc/.

[17] L. Freitas. JACK – A Process Algebra Implementation in Java. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Brazil, April 2002.

[18] L. Freitas. *Model checking Circus*. PhD thesis, Department of Computer Science, The University of York, October 2005.

[19] A. J. 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, August 1996.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[22] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[23] M. E. Nordberg III. Default and Extrinsic Visitor. In *Pattern Languages of Program Design 3*, pages 105–123. Addison-Wesley, 1997.

[24] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[25] P. Malik and M. Utting. CZT: A Framework for Z Tools. In *5th International Conference of Z and B Users (ZB 2005)*, pages 65–84. Springer-Verlag, 2005.

[26] R. C. Martin. Acyclic Visitor. In *Pattern Languages of Program Design 3*, pages 93–103. Addison-Wesley, 1997.

[27] A. McEwan. A Calculated Implementation of a Control System. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures*, pages 265–280. IOS Press, 2004.

[28] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[29] M. V. M. Oliveira. *A Refinement Calculus for Circus*. PhD thesis, Department of Computer Science, The University of York, December 2005.

[30] M. V. M. Oliveira. Unifying Theories of Programming in ProofPower-Z. Technical report, Department of Computer Science, University of York, January 2005. At http://www.cs.york.ac.uk/~marcel/circus/pp/report.pdf.

[31] M. V. M. Oliveira and A. L. C. Cavalcanti. From *Circus* to JCSP. In J. Davies et al., editor, *6th International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, pages 320 – 340. Springer-Verlag, 2004.

[32] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining Industrial Scale Systems in *Circus*. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures*, pages 281 – 309, 2004.

[33] Z Standards Panel. Formal Specificationn – Z Notation – Syntax, Type and Semantics — Consensus Working Draft 2.6, August 2000. At http://www.cs.york.ac.uk/ ian/zstan/.

[34] J. D. Phillips and G. S. Stiles. An Automatic Translation of CSP to Handel-C. In I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch, editors, *Communicating Process Architectures*, pages 19–38, 2004.

[35] V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In J. F. Broenink and G. H. Hilderink, editors, *Communicating Process Architectures*, pages 63–81, 2003.

[36] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[37] S. Schneider and H. Treharne. Verifying Controlled Components. In *4th International Conference on Integrated Formal Methods (IFM 2004)*, pages 87–107, 2004.

[38] A. Sherif and J. He. Towards a Time Model for *Circus*. In H. Miao C. George, editor, *4th International Conference on Formal Engineering Methods (ICFEM 2002)*, volume 2495 of *LNCS*, pages 613–624. Springer-Verlag, 2002.

[39] X. Tang and J. C. P. Woodcock. Towards Mobile Processes in Unifying Theories. In J. R. Cuellar and Z. Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 310 – 319. IEEE Computer Society Press, 2004.

[40] P. H. Welch. Process Oriented Design for Java: Concurrency for All. In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, volume 1, pages 51–57. CSREA Press, 2000.

[41] P. H. Welch and J. M. R. Martin. Formal Analysis of Concurrent Java Systems. In P. H. Welch and A. W. P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pages 275–301. WoTUG, IOS Press (Amsterdam), September 2000.

[42] J. C. P. Woodcock. Using *Circus* for Safety-Critical Applications. In *VI Brazilian Workshop on Formal Methods*, pages 1–15, Campina Grande, Brazil, October 2003.

[43] J. C. P. Woodcock and A. L. C. Cavalcanti. *Circus*: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, July 2001.

[44] J. C. P. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof.* Prentice-Hall, 1996.

[45] M. Xavier. Definição Formal e Implementação do Sistema de Tipos para a Linguagem *Circus*. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Brazil, May 2006. To be submitted.